SECURING ACCESS IN EMBEDDED SYSTEMS VIA DMA PROTECTION AND LIGHT-WEIGHT CRYPTOGRAPHY

by

FILIPPOS – GEORGE KOLYMPIANAKIS

THESIS

submitted in partial fulfillment of the requirements for the

degree MASTER OF SCIENCE

Hellenic Mediterranean University

Department of Electrical and Computer Engineering

Approved by

Supervisor:
Dr Kornaros George

Abstract

Embedded systems are the driving force for technological development in many domains such as automotive, healthcare and industrial control. Security is an important aspect of embedded system design. As more and more computational and networked devices are integrated into all aspects of our lives security becomes critical for the dependability of all smart or intelligent systems built upon these embedded systems. Security is provided through a DMA controller which is operated under certain constraints and with the use of light-weight cryptography as an extended mechanism to safeguard the confidentiality and integrity of stored and transmitted information.

Direct memory access (DMA) protection is a necessity, especially in case of RAM memories where the most of accessible data are located. Different devices or users have different rights to data contained in each memory, which is the main reason to use a firewall as a method to protect the data being accessed. Penetrating a device from high speed ports that permit DMA is an important issue in embedded systems. A DMA firewall prevents physical connections from DMA attacks so that each device has restricted access for DMA transfers using memory limitations and device ID. In that way attackers are prevented from stealing data or cryptographic keys, install or run spyware and other exploits or modify the system to allow backdoors or other malware

Table of Contents

Contents

List of Figures

Acknowledgements

This thesis was prepared by the postgraduate student Kolimpianakis Filippos – George under the supervision of Professor George Kornaros. To Dr. Kornaros George I owe my sincere gratitude for his guidance and support throughout the processing of this Graduate Thesis.

1. Introduction

An embedded system is a computer system with a dedicated function within a larger mechanical or electrical system, often with real-time computing constraints [54]. Embedded systems control many devices in common use today and ninety-eight percent of all microprocessors are manufactured as components of embedded systems [54]. Applications of embedded systems include smartphones, mp3 players, video game consoles, digital cameras, GPS receivers and technically every device that takes an input and can produce an output and functions as an independent system [54].

Direct Memory Access (DMA) is feature of computer hardware that allows devices to gain access to the main bus linking the processor to the system memory and move data directly between the main memory and another part of the system [57]. A dedicated DMA controller, often integrated in the processor, can be configured to move data between main memory and a range of subsystems, including another part of main memory [57].

Security and trusted operation of cyberphysical systems [59][64][65] is increasingly an important concern in modern digitalized, software-dominated internet-of-everything. Various techniques have been proposed to guarantee security for embedded devices, which focus to automotive systems and network communications [58][60][61][63] but also to protect internal execution of applications from faulty or malicious behavior [66][62].

1.1 General

Many embedded systems have internal and external interfaces that produce or consume data [57]. These can be simple UARTs, external bus devices or complicated video and graphics devices [57]. A key part of any embedded system is ensuring that data flowing in to and out of these interfaces is handled properly and not lost or corrupted [57]. A simple way of moving the data between the peripheral device and main memory is to use the main processor to perform load or store operations for each byte or word of data to be moved [57]. The processor must wait for the peripheral to be ready before transferring each byte or word [57]. For many systems this is not a good use of processing time as the processor may be spending more time than necessary moving data between main memory and its external devices [57]. The alternative way is to set up a DMA transfer that gives the job of moving data to a special-purpose device in the system [57]. Once the processor has set up the transfer it can be occupied in his main task while the transfer is in progress or wait to be notified when the transfer has finished [57].

The obvious benefit of moving data using DMA transfers is that the processor can do something else while the transfer is in progress. However, using DMA sometimes has other advantages depending on the hardware involved [57]. These include:

Data transformations – applications targeted to video or digital signal processing, may be able to perform data transformations as part of the DMA transfer. These include byte-order changes and 2D block transfers [57]

Lower power – if the processor load is reduced and there are fewer interrupts it may be possible to run the processor at a lower clock rate or even to enter a low power mode while DMA transfers are in order [57].

Higher data throughput – a given processor may be able to handle more external interfaces at higher data rates, or a low-end processor might be able to handle more complicated interfaces such as Ethernet or USB [57].

The simplest is a known as a single-cycle DMA transfer and is typically used to transfer data between devices such as UARTs or audio codecs that produce or consume data a word at a time [57]. In this situation the peripheral device uses a control line to signal that it has data to transfer or requires new data. The DMA controller obtains access to the system bus, transfers the data, and then releases the bus. Access to the bus is granted when the processor, or another bus master, is not using the bus. [57].

Another type of transfer is a burst transfer. This is used to transfer a block of data in a series to the system bus. The transfer starts with a bus request; when this is granted, the data is transferred in bursts [57]. The burst size depends on the processor architecture and the peripheral, and may be programmable depending on the details of the hardware [57]. The last mode of operation is the transparent mode where the CPU never stops to execute its programs and the DMA controller transfer data free, but those programs do not use the system buses, so each party works on its own [68].

While a burst transaction is occurring, the processor will not be able to access the system bus. However, preventing the processor from accessing the system bus may cause a delay, which can reduce the system performance [57]. To minimize the effects of this problem, the DMA controller may release the bus after a fixed number of burst transactions or when a pre-determined bandwidth limit has been reached. However, if the system needs to perform large DMA block transfers the system designer needs to carefully work out the bus bandwidth requirements to ensure there are no performance bottlenecks in the hardware or software design [57].

Before a DMA transfer can begin the processor must address the DMA controller the amount of data and the location to be moved. A DMA transfer usually has these attributes:

Source address - the address from where the data is transferred, in main memory or the peripheral address space

Destination address - the address to where the data is transferred, in main memory or the peripheral address space

Transfer length - the overall length of the transfer, specified in terms of bytes or words.

Embedded system security is the reduction of vulnerabilities and protection against threats in software running on embedded devices. Like security in most IT fields, embedded system security involves a conscientious approach to hardware design and coding as well as added security software, an adherence to best practices and consultation with experts [55]. A DMA attack is a type of side channel attack in computer security, in which an attacker can penetrate a computer or other device, by exploiting the presence of high-speed expansion ports that permit direct memory access (DMA) [55].

Cryptography is about constructing and analyzing protocols that prevent third parties from reading private messages [56]. Various aspects in information security such as data confidentiality, data integrity, authentication, and non-repudiation are central to modern cryptography [56]. Modern cryptography exists at the intersection of the disciplines of mathematics, computer science, electrical engineering, communication science, and physics. Applications of cryptography include electronic commerce, chip-based payment cards, digital currencies, computer passwords, and military communications [56].

1.2 Motivation

An embedded system controlling its peripheral devices and interfaces with the use of identification and permission-controlled DMA can provide the secure and reliable communication between memory and those peripheral devices. In addition, the use of light weight cryptography mechanism provides further more protection in a low-cost design and high-speed communication. One high performance DMA module with multiple channels and high throughput should provide the security needed through multiple AES hardware blocks designed and developed as a part of this thesis.

1.3 Contribution

The main contribution of this thesis is to design a DMA controller with permission rights for each device connected using embedded AES cryptographic hardware system for the transmitted data. As it will be demonstrated in the following sections, we develop such a system which provides a simple but reliable function in transmitting data between memory and peripheral devices in a high speed and low-cost design.

A device such as this can be used in a wide area of application. In medical applications for online patient monitoring (sensitive data and who can access it), in automotive for firmware update and memory data protection, smartphone or tablet memory protection mainly when accessing the internet, updating or installing new application, in surveillance system with camcorders with the use of id for data protection, in wave generation on real time operating systems and generally prevent malware attack. Those are some of the possibilities a system like this has to offer.

1.4 Organization

The general approach of this work is to establish a secure protected DMA transmission between memory and peripheral devices on an embedded system with the use of hardware AES cryptography blocks. Both DMA controller and AES blocks are designed and developed as part of the work of this thesis. The remainder of this thesis is organized as follows:

- Chapter 2 states some Basic Principles
- Chapter 3 examines Embedded Systems Security
- Chapter 4 introduces a background on related work
- Chapter 5 analyzes the architecture and design methodology of the hardware and software infrastructure that has been developed
- Chapter 6 presents the performance results and analysis of the system developed as well as the comparison with the systems existing
- Chapter 7 concludes with the summary of this work and directions for future extensions

2. Basic Principles

Information security is the practice of protecting information and preventing unauthorized/inappropriate access, use, disclosure, disruption, deletion/destruction, corruption, modification, inspection, recording or devaluation of information, or at least reducing the chance of that happening [67]. The idea is to assure the authenticity, integrity, availability, confidentiality, non-repudiation of the data transmitted either when referring on Personal Computers, or in this case on Embedded systems [67] [3]. Security of embedded systems is creating a lot of issues because of poor security design as well as implementation and resource constraint [41].

Protection in Embedded systems is an important factor when in stage of designing of such systems. As explained in next chapter embedded systems like every device in personal computing has a lot of value Constrains, Vulnerabilities and Attacks, so in the stage of designing there has to be serious consideration in the methods to be used, cost of the embedded device as well as to the performance of the system expected. As observed in many cases the most of the attacks on an embedded device was targeting the DMA controller as the most data transfers origin from the specified controller. Some basic principles and architectures will be described in the sections following below.

2.1 Embedded Systems

Features demanded in todays embedded systems include acceleration, flexibility, personalization, security, privacy, redundancy, scalability, modularity, root-of-trust, PUF-based keys, longer key sizes, etc. and especially in the case of multi-processor SoC (MPSoC) platforms and FPGAs [5]. Most of the SoCs nowadays use a security co-processor designed in register transistor level (RTL) on embedded systems developed for Smart grid uses sensors, monitoring, communications, automation and computers [6] A common application of coprocessors is the acceleration of cryptographic algorithms and gain speed. The purpose of designing such platforms is to improve the flexibility, security, reliability, efficiency, and safety of those systems [6]. The main idea when developing embedded systems is to achieve lower power consumption and lower design or manufacturing cost as much as possible but maintain user friendliness, feasibility and expandability [6].

Embedded devices transmit a large amount of sensitive data that include bank account numbers, passwords, social security numbers, medical records, etc. [7]. An attacker can develop malicious applications, thus, when installed on the device can obtain access to private sensitive information [7]. The best way to prevent these applications is to use anti-virus, anti-malware, and anti-spyware software which require a high amount of computational power and resources [7]; thus, the best way is to use cryptographic algorithms to protect sensitive data on SoC devices used in healthcare, home automation or automotive industry [13]. Embedded systems in automotive, healthcare and industrial control are progressively getting computationally efficient and network enabled, thus, security is becoming crucial for smart and intelligent systems built in these embedded systems [30] [41].

2.2 Direct Memory Access

Direct Memory Access (DMA) is an attribute of Computer Systems for memory access and data transfer between memory and peripherals without the use of CPU [68]. The aim of DMA is to remove the load of the CPU so that it can be occupied with other operations. The processor initiates the DMA controller by sending the source address, Number of words and the destination address of data [53] [68]. Main advantages and modes of operation have been described in Chapter one so they will not be discussed further.

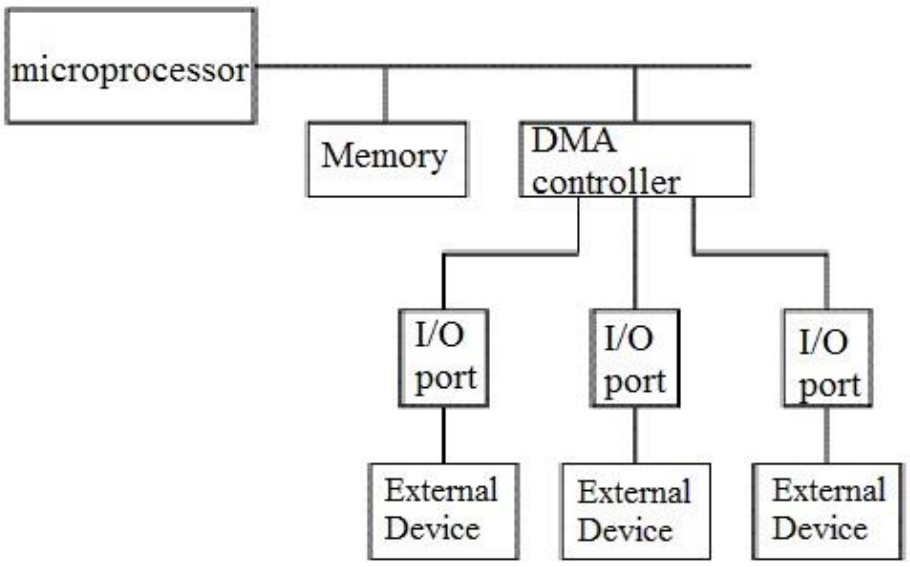


Figure 1: Typical DMA

As the use of DMA has become well known so has the large number of attacks on those controllers [69]. Direct memory access (DMA) protection is a necessity, especially in case of RAM memories where the most of accessible data that are currently being used by each device are located [69]. Penetrating a device from high speed ports that permit DMA is an important issue in embedded systems. Attackers use these ports to gain access to physical memory address space, acquire the devices purpose, steal data or cryptographic keys, install or run spyware and exploits or modify the system to allow backdoors or other malware [69].

2.3 Cryptography

Modern cryptography depending on the key can be either symmetric (private key cryptography) or asymmetric (public key cryptography) [56]. In Symmetric-key cryptography both sender and receiver share the same key either if referring to Block ciphers, where inputs are in blocks, or stream ciphers, where inputs are individual characters [56]. Encryption and Decryption on symmetric algorithms use the same private key in each method. Most common private key algorithms are DES, 3-DES and AES (which is the one to be implemented in this thesis) [56]. In Asymmetric-key cryptography, sender and receiver use a pair of keys, a public key and a private key. Public-key algorithms are more computationally demanding because of the level of security they provide but are ideal for digital signature [56]. Most common algorithms are RSA, DSA and lately ECC (These is not going to be further reference thus it is not the topic of this thesis).

2.3.1 Advanced Encryption Standard

The Advanced Encryption Standard (AES) originally known as Rijndael from its developers (Vincent Rijmen and Joan Daemen) is a symmetric cryptography algorithm (private key) block cipher with key sizes 128, 192 and 256 bits, so the same key is used for encryption and decryption [70]. AES operates on a two-dimensional table (4 x 4 array) of bytes called "*state*" and it takes 10 transformation rounds to convert the input, who is called plaintext, to the final output, who is called ciphertext, for a 128-bit key size, 12 rounds for 192-bit key size and 14 rounds for 256-bit key size [70].

There are four steps of operation in each round of processing [70]. The first step is called "*SubBytes*" in which a 16 x 16 lookup table (substitution box or sbox) is used in order to find a replacement byte for every given byte of input [70]. In the second step called "*ShiftRows*" the rows of the table shift to left by zero, one, two and three according to the row [70]. The third step is "*MixColumns*" where each byte of a column is replaced of all the bytes in the same column, in particularly each byte in a column is replaced by two times that byte, plus three times the next byte, plus the byte that comes next, plus the byte that follows (the simplest way to explain the step) [70]. The final step is called "*AddRoundKey*" in which the round key is added to the output with the XOR logical operation [70]. The round key is computed in every round in which every Byte is replaced with another using the lookup table and the xor logical operation [70].

The four steps (which can be seen in figures below) explained in previous paragraph are executed in the order presented in every transformation round, except the last transformation round in which the “*MixColumns*” is not executed. This operation is for the encrypt mode of execution, during decryption mode inversive steps are executed in reverse mode of operation as it can be seen in detail in the figure below.

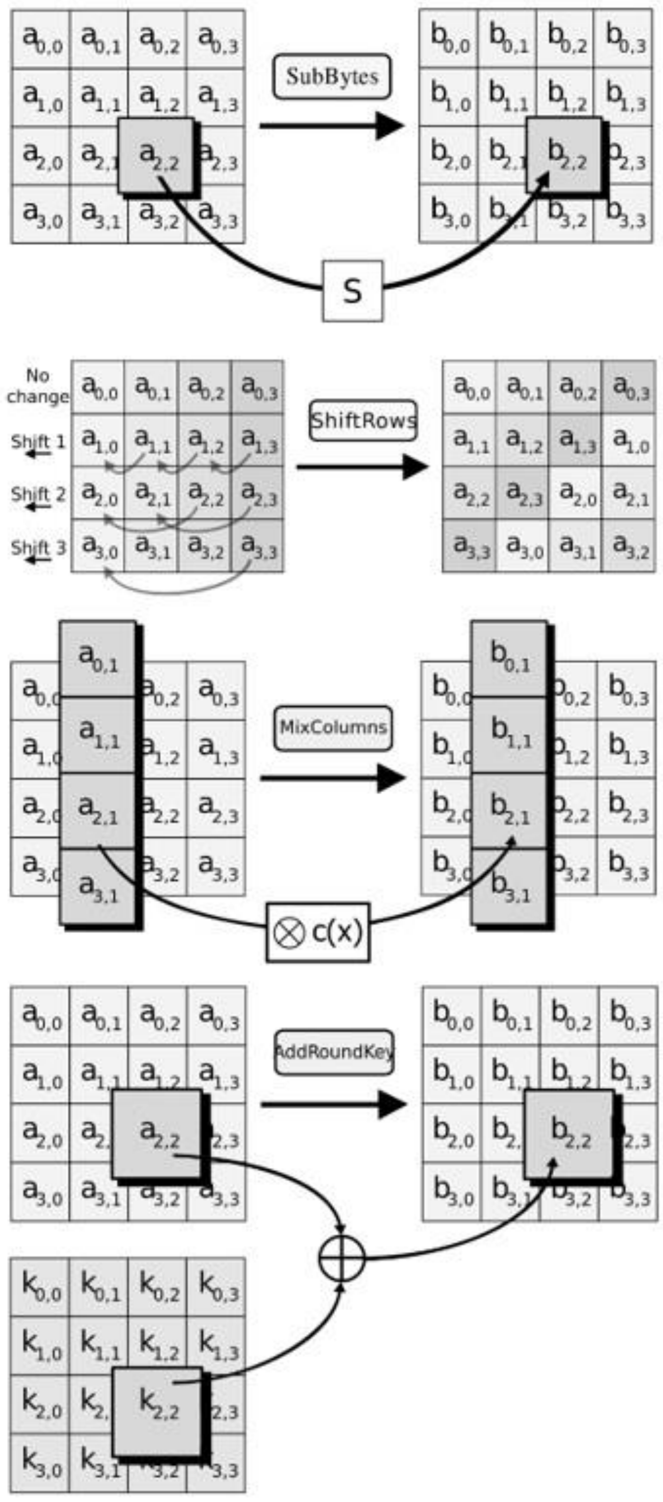


Figure 2: Steps of AES Encryption (Wikipedia)

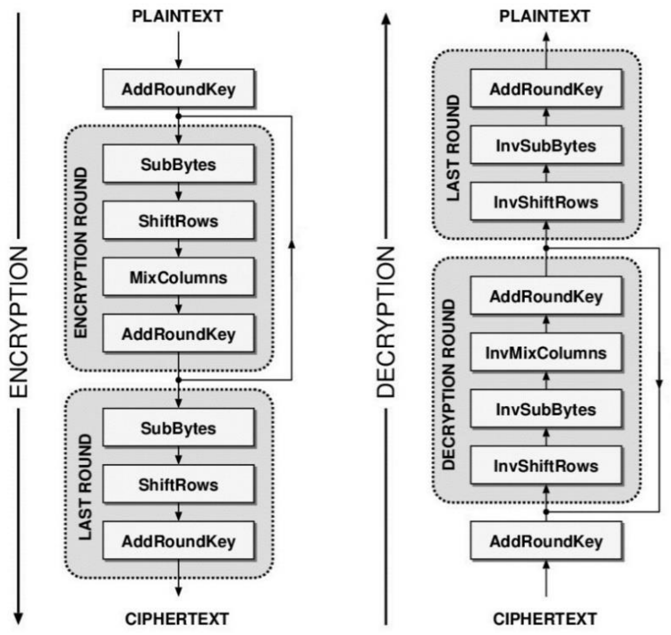


Figure 3: AES algorithm

The “S-box substitution” is a non-linear transformation of two steps. First, the input bytes are multiplicative inversed and second, a transformation is applied [38]. S-box entries are implemented using look-up tables or computed mathematically [38]. Transformations “Shift-Rows” and “Mix-Columns” are linear operations [38]. The AES is a very widespread symmetric cryptography algorithm for encrypting data [38]. The AES algorithm has two modes of operation, the Electronic Codebook (ECB) mode and the Cipher Blocker Chaining (CBC) mode. The ECB mode in general is the main generation of AES in which each plaintext is encrypted as an independent block. In CBC mode each plaintext is XORed with the previous ciphertext so every block is dependent on the previous, thus, providing a higher level of security but consumes more time. The CBC mode was created because in ECB mode identical blocks produce the same ciphertexts, thus, it turns out to be a security risk. FPGAs in general offer the performance required to implement such an algorithm [38].

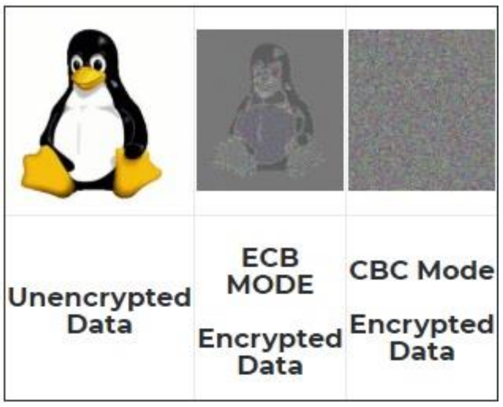


Figure 4: ECB vs CBC (Wikipedia)

2.3.2 True Random Number Generator

For key generation often as a safe way to generate keys between parties often is used a hardware random number generator (HRNG) or true random number generator (TRNG). A TRNG is a device that generates random cryptographic keys to transmit data securely [71]. Most of the related work in the past use a TRNG to produce cryptographic keys [71].

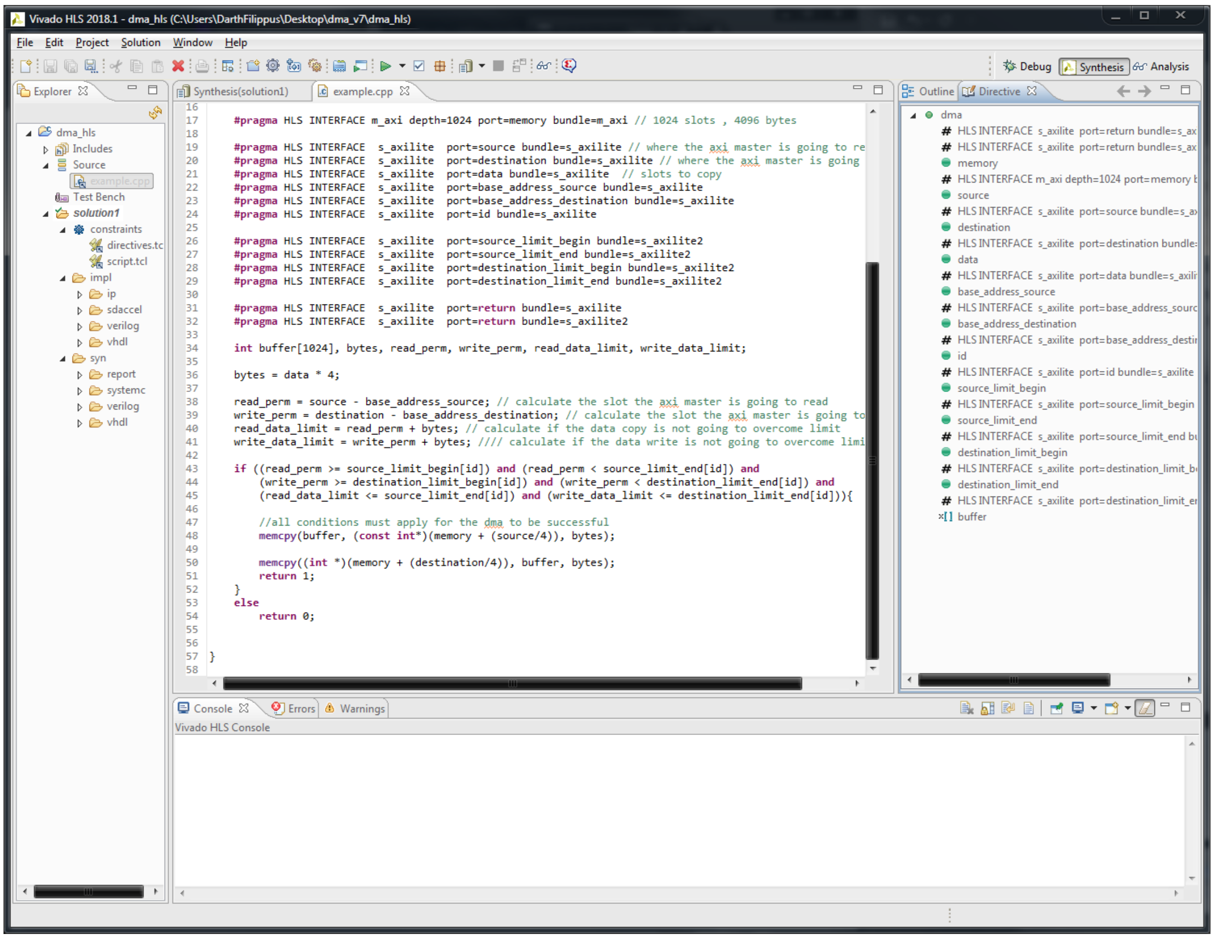
2.4 Xilinx Vivado

Vivado Design Suite is a development environment from Xilinx for system-level integration and implementation in order to design IP-centric SoCs. This software suite is produced for synthesis and analysis for hardware description language designs with a wide area of features including system on chip development and high-level synthesis analysis [72]. It is better conception compared to Xilinx ISE development kit, faster and more integrated [72]. This software suite is used in order to develop practical part of this thesis. The software consists of Vivado High-Level Synthesis (HLS), the Vivado Software and the Software Development Kit (SDK).

It is important to indicate that the communication protocol used between cores is the AXI4 interface protocol. Advanced eXtensible Interface (AXI) is a part of the ARM Advanced Microcontroller Bus Architecture (AMBA) with a parallel high-performance, synchronous, high-frequency, multi-master, multi-slave communication interface, mainly designed for on-chip communication [81]. AXI4 is the interface protocol used in the architecture of this project for all parties communication, from Microprocessors, IP blocks to peripheral devices and external ports.

2.4.1 Vivado High Level Synthesis

Vivado HLS is a software tool included in the Vivado Development kit that accelerates IP creation to be used in SoCs. The idea is to create hardware modules in high level programming language like C, C++ and SystemC and the software after a successful synthesis exports RTL. The IP block exported is ready to be used in Vivado or ISE and can be used and reused many times. The software window is shown in the figure below.



```
    #pragma HLS INTERFACE m_axi depth=1024 port=memory bundle=m_axi // 1024 slots , 4096 bytes

    #pragma HLS INTERFACE  s_axilite  port=source bundle=s_axilite // where the axi master is going to re
    #pragma HLS INTERFACE  s_axilite  port=destination bundle=s_axilite // where the axi master is going
    #pragma HLS INTERFACE  s_axilite  port=data bundle=s_axilite  // slots to copy
    #pragma HLS INTERFACE  s_axilite  port=base_address_source bundle=s_axilite
    #pragma HLS INTERFACE  s_axilite  port=base_address_destination bundle=s_axilite
    #pragma HLS INTERFACE  s_axilite  port=id bundle=s_axilite

    #pragma HLS INTERFACE  s_axilite  port=source_limit_begin bundle=s_axilite2
    #pragma HLS INTERFACE  s_axilite  port=source_limit_end bundle=s_axilite2
    #pragma HLS INTERFACE  s_axilite  port=destination_limit_begin bundle=s_axilite2
    #pragma HLS INTERFACE  s_axilite  port=destination_limit_end bundle=s_axilite2

    #pragma HLS INTERFACE  s_axilite  port=return bundle=s_axilite
    #pragma HLS INTERFACE  s_axilite  port=return bundle=s_axilite2

    int buffer[1024], bytes, read_perm, write_perm, read_data_limit, write_data_limit;

    bytes = data * 4;

    read_perm = source - base_address_source; // calculate the slot the axi master is going to read
    write_perm = destination - base_address_destination; // calculate the slot the axi master is going to
    read_data_limit = read_perm + bytes; // calculate if the data copy is not going to overcome limit
    write_data_limit = write_perm + bytes; //// calculate if the data write is not going to overcome limi

    if ((read_perm >= source_limit_begin[id]) and (read_perm < source_limit_end[id]) and
        (write_perm >= destination_limit_begin[id]) and (write_perm < destination_limit_end[id]) and
        (read_data_limit <= source_limit_end[id]) and (write_data_limit <= destination_limit_end[id])){

        //all conditions must apply for the dma to be successful
        memcpy(buffer, (const int*)(memory + (source/4)), bytes);

        memcpy((int *)(memory + (destination/4)), buffer, bytes);
        return 1;
    }
    else
        return 0;


}
```

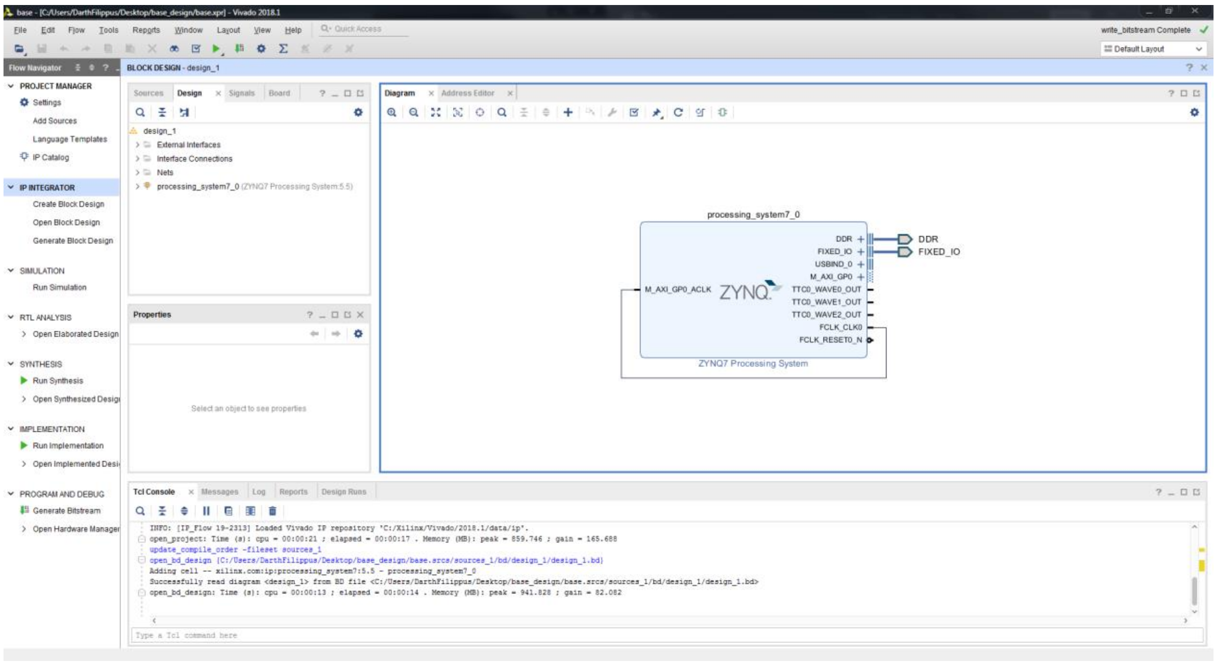
```
dma
  # HLS INTERFACE  s_axilite  port=return bundle=s_ax
  # HLS INTERFACE  s_axilite  port=return bundle=s_ax
  memory
  # HLS INTERFACE m_axi depth=1024 port=memory b
  source
  # HLS INTERFACE  s_axilite  port=source bundle=s_ax
  destination
  # HLS INTERFACE  s_axilite  port=destination bundle=
  data
  # HLS INTERFACE  s_axilite  port=data bundle=s_axili
  base_address_source
  # HLS INTERFACE  s_axilite  port=base_address_sourc
  base_address_destination
  # HLS INTERFACE  s_axilite  port=base_address_destir
  id
  # HLS INTERFACE  s_axilite  port=id bundle=s_axilite
  source_limit_begin
  # HLS INTERFACE  s_axilite  port=source_limit_begin
  source_limit_end
  # HLS INTERFACE  s_axilite  port=source_limit_end bu
  destination_limit_begin
  # HLS INTERFACE  s_axilite  port=destination_limit_b
  destination_limit_end
  # HLS INTERFACE  s_axilite  port=destination_limit_er
  x[] buffer
```

Figure 5: Vivado HLS

2.4.2 Vivado

In the Vivado Software, IP Blocks can be added and connected in the AXI -4 interface. IP Blocks from the Vivado repository include processing modules, memories, interconnects, DSP systems, mathematical functions, bus interfaces, network functionalities and many more as well as IP Blocks created in Vivado HLS can be imported and used at will. After a design is validated by the software in can be synthesized. The synthesis function is the similar function as the compile function but in this case the Block Design is synthesized in logical ports that it will use in implementation.

After a successful synthesis follows the implementation stage where logical ports from the synthesis design are translated in an FPGA design in order to download it on a FPGA platform. It also places and routes the logical ports into device resources. In this case the FPGA platform to be used is the ZedBoard described in the next section. If the implementation is successful the bitstream file can be generated in order to program the FPGA target device. The software window can be seen in the section below.

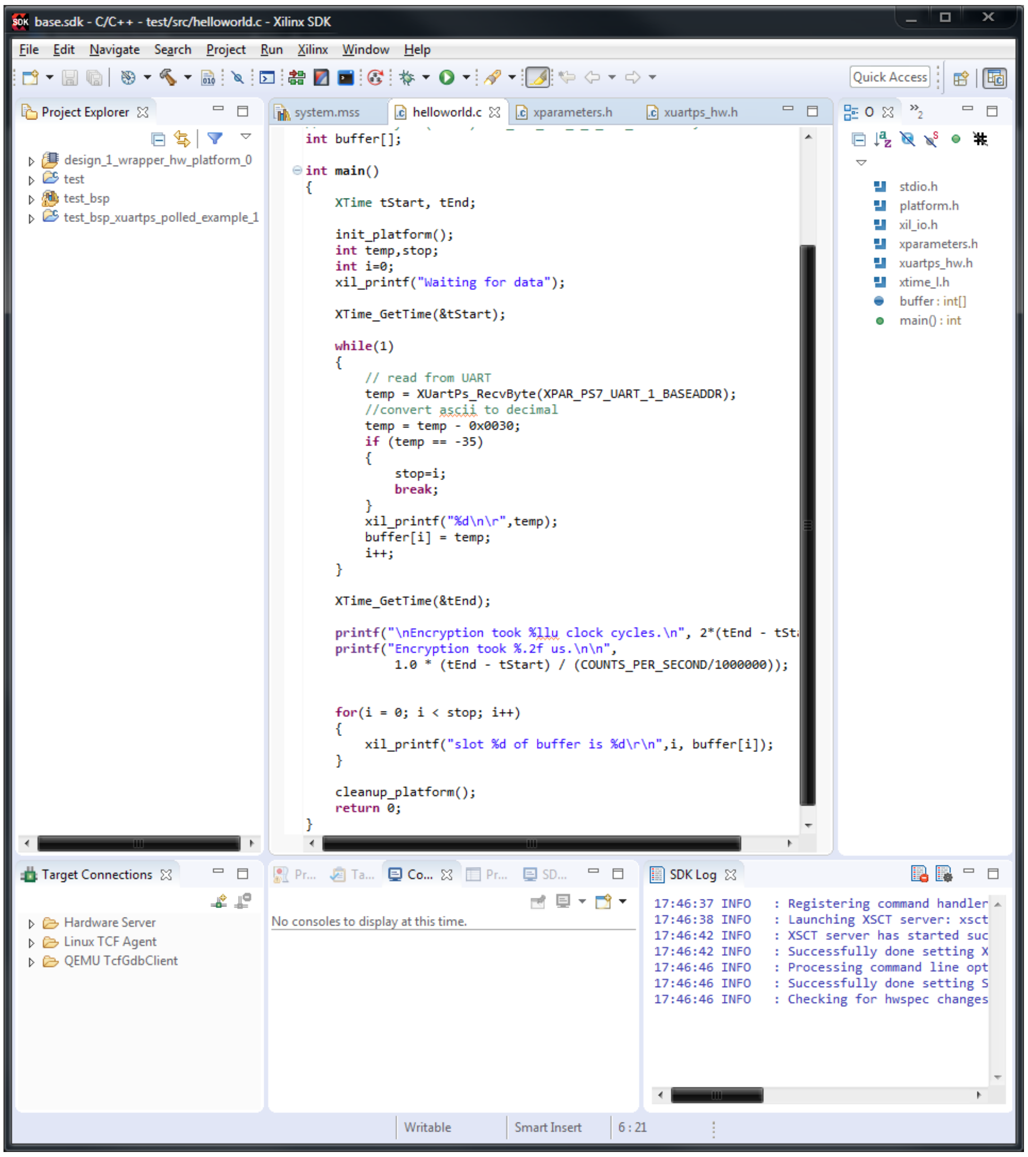


```
INFO: [IP_Flow 19-2313] Loaded Vivado IP repository 'C:/Xilinx/Vivado/2018.1/data/ip'.
open_project: Time (s): cpu = 00:00:21 ; elapsed = 00:00:17 . Memory (MB): peak = 859.746 ; gain = 165.688
update_compile_order -fileset sources_1
open_bd_design {C:/Users/DarthFilippus/Desktop/base_design/base.srcs/sources_1/bd/design_1/design_1.bd}
Adding cell -- xilinx.com:ip:processing_system7:5.5 - processing_system7_0
Successfully read diagram <design_1> from BD file <C:/Users/DarthFilippus/Desktop/base_design/base.srcs/sources_1/bd/design_1/design_1.bd>
open_bd_design: Time (s): cpu = 00:00:13 ; elapsed = 00:00:14 . Memory (MB): peak = 941.828 ; gain = 82.082
```

Figure 6: Vivado Design Software

2.4.3 Software Development Kit

The Xilinx Software Development Kit (SDK) is a software environment for creating applications for embedded systems on microprocessors like Zynq UltraScale+ MPSoC, Zynq-7000 SoCs and Microblaze. The SDK application offers multiprocessors designs, debugger and performance analysis. On this software program the applications are developed that is going to be run on the embedded device. Through this program the FPGA of the hardware target can be programmed and run the applications that are developed on the FPGA or the embedded microprocessor. The program is included in the Xilinx Vivado package and can be seen in the figure below.



```
int buffer[];

int main()
{
    XTime tStart, tEnd;

    init_platform();
    int temp,stop;
    int i=0;
    xil_printf("Waiting for data");

    XTime_GetTime(&tStart);

    while(1)
    {
        // read from UART
        temp = XUartPs_RecvByte(XPAR_PS7_UART_1_BASEADDR);
        //convert ascii to decimal
        temp = temp - 0x0030;
        if (temp == -35)
        {
            stop=i;
            break;
        }
        xil_printf("%d\n\r",temp);
        buffer[i] = temp;
        i++;
    }

    XTime_GetTime(&tEnd);

    printf("\nEncryption took %llu clock cycles.\n", 2*(tEnd - tSt
    printf("Encryption took %.2f us.\n\n",
           1.0 * (tEnd - tStart) / (COUNTS_PER_SECOND/1000000));


    for(i = 0; i < stop; i++)
    {
        xil_printf("slot %d of buffer is %d\r\n",i, buffer[i]);
    }

    cleanup_platform();
    return 0;
}
```

```
No consoles to display at this time.
```

```
17:46:37 INFO   : Registering command handler
17:46:38 INFO   : Launching XSCT server: xsct
17:46:42 INFO   : XSCT server has started suc
17:46:42 INFO   : Successfully done setting X
17:46:46 INFO   : Processing command line opt
17:46:46 INFO   : Successfully done setting S
17:46:46 INFO   : Checking for hwspec changes
```

Figure 7: Software Devepopment Kit

2.5 ZedBoard

ZedBoard is a full development kit from AVNET that uses the Xilinx Zynq-7000 All Programmable SoC and supports a large area of applications. This is the FPGA hardware platform that is going to be used for this project so some of his features has to be mentioned. Features of the Zedboard include the Zynq – 7000 SoC processing system with ARM dual core processor A9 (677MHz max Frequency), 512 MB DDR3, 256 Mb Quad-SPI Flash, 4 GB SD card, Onboard USB-JTAG Programming 10/100/1000 Ethernet, USB OTG 2.0 and USB-UART, FMC-LPC connector (68 single-ended or 34 differential I/Os), 5 Pmod™ compatible headers (2x6), Agile Mixed Signaling (AMS) header, 33.33333 MHz clock source for PS, 100 MHz oscillator for PL, HDMI output supporting 1080p60 with 16-bit, YCbCr, 4:2:2 mode color, VGA output (12-bit resolution color), 128x32 OLED display, Onboard USB-JTAG interface, Xilinx Platform Cable JTAG connector, 8 user LEDs, 7 push buttons and 8 DIP switches. ZedBoard can be seen in the figure below.

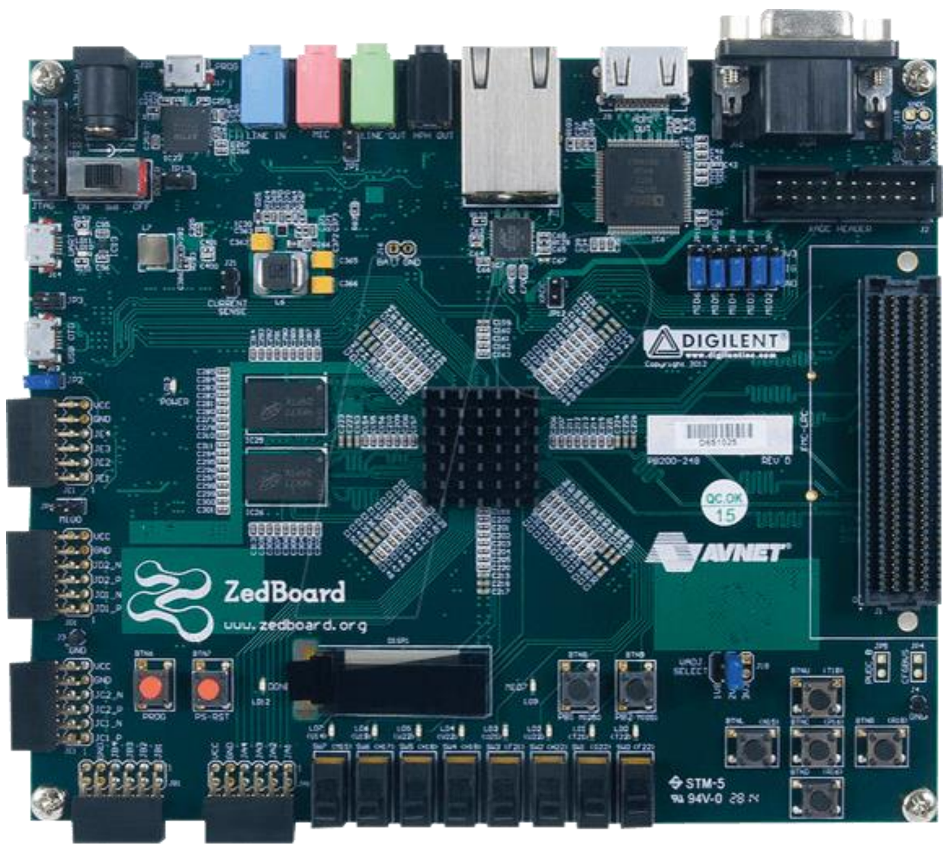


Figure 8: Zedboard

3. Embedded Systems Security

Application of embedded systems can be found in in Automobiles, in Telecommunication to Motor and cruise control system, for Body or Engine safety, for Entertainment and multimedia in car, to E-Com and Mobile access, for Robotics in assembly line or Wireless communication [28]. Also, in Smart Cards, Missiles and Satellites for Security systems, Telephone and banking, Defense and aerospace or Communication [28]. In Peripherals and Computer Networking for Displays and Monitors, Networking Systems, Image Processing, Network cards and printers and in Consumer Electronics on Digital Cameras, Set top Boxes, High Definition TVs, DVDs [28]. All these applications have some constrains, vulnerabilities and attacks which are discussed in this chapter.

Most of the application above use a high-end processor for some real-time systems and integrate a DMA peripheral to handle the data transfers and the bus access [43]. The DMA reduces the power consumption it increases the CPU’s bandwidth and frees up available processing capacity but more over the features of DMA are: a)Access to multiple peripherals in the processor, b)Multiple DMA channels in the processor, c)Different priorities for each channel, d)Handling the source and destination address, e)Handling the increment/ decrement of the source and destination address based on the configuration, f)Checking for the completion of the transfer of configured number of bytes for a DMA channel, g)Burst transfers which split the whole data transfer into multiple blocks and h)Generate an interrupt when required [44].

When designing secure embedded systems some factors has to be taken into consideration such as small form factor, good performance, low energy consumption (and, thus, longer battery life), and robustness to attacks [26]. Such attacks can include viruses, malwares, worms, physical tampering, and side-channel attacks [26]. Security engineering when developing intelligent devices like wireless phones, routers and switches, printers, SCADA (Supervisory Control and Data Acquisition) systems, electronic devices in cars and even medical devices has been an important aspect when users tend to manipulate them or harm then in any way [26]. Mileage counter manipulation, unauthorized chip tuning or tachometer spoofing has forever been a security risk [26]. Encryption has forever been computationally intensive and requires dedicated hardware in an environment where resources of the embedded system are restricted [26]. The design must meet the arrangements and the capacity as well as the cost of the embedded system. As attacks continue to increase, the development of countermeasures and the insurance of a secure execution environment (SEE) as well as the dynamic adaptation has to be a must have in embedded system security [26].

3.1 Constrains, Vulnerabilities and Attacks

There have been observed and categorized 60,000 entries of Common Vulnerabilities and Exposures according to the survey in [29]. From these entries the authors have classified the attacks concerning embedded systems. In precondition class the requirements of the attacker include Internet facing device, Local or remote access to the device, Direct physical access to the device, Physically proximity of the attacker, Miscellaneous or Unknown [29]. For vulnerability they found Programming errors, Web based vulnerability, Weak access control or authentication, Improper use of cryptography and Unknown [29]. As far as the target concerned the hardware, firmware or OS and application [29]. For the attack methods they described Control hijacking attacks, Reverse engineering, Malware, Injecting crafted packets or input, Eavesdropping, Brute-force search attacks, Normal use and Unknown [29]. Last, as the effect of the attack they found Denial-of-Service, Code execution, Integrity violation, Information leakage, Illegitimate access, Financial loss, Degraded level of protection, Miscellaneous and Unknown [29].

On the overview of [30] the authors categorize the topics of embedded systems security. Some disadvantages in the nature of embedded systems include Limited processing power, Limited available power, Physical exposure, Remote and unmanned operation and limited network connectivity [30]. Some vulnerabilities Energy drainage (exhaustion attack), Physical intrusion (tampering), Network intrusion (malware attack), Information theft (privacy), Introduction of forged information (authenticity), Confusing/damaging of sensor or other peripherals, Thermal event (thermal virus or cooling system failure), Reprogramming of systems for other purposes (stealing) [30]. Typical attacks target at privacy authenticity, access control and confidentiality. The attacks can be physical, logical or side channel based. physical attacks include micro probing, reverse engineering and eavesdropping [30]. On the other hand, logical can be software based or cryptographic [30]. Some countermeasures against software attacks are based on architecture, on safe languages, static code analyzers, dynamic code analyzers, anomaly detection techniques, sandboxing or damage containment approaches, compiler support, library support and based on Operating system [30]. Countermeasures against side channel attacks are masking, window method, dummy instruction insertion, code/algorithm modification, balancing, etc. [30].

Survey [7] mentions existing defenses, both software and hardware based such as Software-based Mechanisms, watchdog checkers, Merkle trees, integrity trees, memory encryption, and modification of processor architecture. The paper categorizes the hardware attacks as Evil maid attack, Cold boot attack, Firewire DMA attack, Bus attack and software attacks as Buffer overflow attacks, Return-into-libc and Code injection attacks [7].

The developers in embedded systems had to use either software-based techniques or a coprocessor to achieve public-key cryptography. Software-based techniques are too computationally expensive for an 8-bit microcontroller. On the other hand, Coprocessor-based implementations need extra cost in hardware to support a single type of public-key cryptography [2]. Solution to that problem is the technique of reusable hardware, the share of computation units and the use of SRAM memory as cache [2]. One problem that arises is the level performance bottlenecks occurring from data transfers between the RAM and the coprocessor [2].

In medical field the physician uses embedded technology to acquire medical information about a patient where security is an important factor considering the doctor patient confidentiality [28]. With the use of embedded devices, the treatment of patients can be faster and more reliable. Application of embedded systems in medical field include Fetal Monitor, Oximeter, Defibrillator, Digital Flow Sensor, Pacemaker, Pulse oximeter, etc [28]. Types of these application can either be real time, standalone, networked or mobile in a small scale, medium scale or sophisticated [28].

IoT Sensor Devices and Automotive Electronic Control Units (ECUs) have a need for lightweight cryptography because they control critical functions as braking, acceleration etc. and are connected on a Controller Area Network (CAN) [40]. Those circuits have a need for Small code/area footprint, minimum latency and low power and that is the reason to use lightweight cryptography [40]. Automotive embedded Systems nowadays allow software installation, remote updates, data sharing, or application input data so software corruption has become a major concern [41].

Other protocols except the CAN in ECUs include Local Interconnect Network (LIN), FlexRay (for better rates of steering wheel and brakes), MOST (for multimedia) and Ethernet [51]. Every attack on these protocols target on Theft, Electronic tuning, Sabotage, Intellectual property theft, Privacy breach and Intellectual challenge [51]. Attacks can be categorized as Internal and Remote. Internal attacks include Vulnerabilities on the bus and Local attacks [51]. Remote include Indirect access (OBD port, CD player, USB port), Short range attacks (Wireless pairing of mobile devices, Car-to-car communications, Tire Pressure Monitoring System, Wireless unlocking), Long-range direct attacks (Telephony, Web browsing) and Long-range indirect attacks (App store, Side channel triggers) [51]. Constraints on automotive embedded systems contain Hardware constrains, real time, Autonomy, Physical constraints, Lifecycle and

Compatibility constrains [51]. It is important when designing a secure communication architecture for internal or intervehicular communication to secure external communication [51]. Internal protections contain Cryptographic solutions, Solutions detecting anomalies in the system and Solutions to ensure integrity of the embedded software [51].

One literature survey [41] refers to attacks in Automotive Embedded systems. Low-level memory such as RAM, FLASH, and CACHEs are very important to ECUs and if an attacker manages to change the firmware, he can do serious damage to the entire system [41]. ECUs control and regulate information acquired by the sensors (Sensors convert physical quantitates to electrical signals and actuators to motion.) and there are 70 ECUs in one automobile with more than 100 million lines of code [41] [51]. Every memory in an ECU (flash and RAM) has a primary bootloader (PBL) and a secondary bootloader (SBL). PBL loads the application software from FLASH and the SBL loads from PBL, so every update that is to be made occur only on SBL [41]. It is important to have restrict access to the bootloader [42].

Features and characteristics of the automotive embedded system include limited processing power, limited power supply, physical exposure and network connectivity as written above in [26], same principles apply here [41]. Threats identified as Energy drainage (exhaustion attack), Physical intrusion (tampering), Network intrusion (malware attack), Introduction of forged information (authenticity), Confusing/damaging of sensor or other peripherals, Reprogramming of systems for other purposes (stealing) [41] [53]. Vulnerabilities in embedded systems have been discussed previously but as far as automotive is concerned are Programming errors, Network based vulnerability, Weak access control or authentication, Improper use of cryptography [29] [41]. Requirement for the attacker are Internet facing device, Local or remote access to the device, Direct physical access to the device, Physically proximity of the attacker and general attacking strategies on memories [30] [41].

Attacks can be classified as physical, logical and side channel attacks [30], [41]. Physical include Reverse engineering, Micro-probing, Eavesdropping and memory access Using Debuggers [30] [41] [42]. Logical refer to Code injection attacks (Stack based buffer overflow, Heap based buffer overflow, Shell code injection, return to libc attack, Return oriented programming), Cryptographic attack (Brute forcing, Dictionary attack) [30] [41] [42]. Last side channel attacks are based on Fault injection attacks, Timing analysis attack, Power analysis attack and DMA attacks [30] [41].

Wireless sensor nodes and RFID tags are very efficient devices that consume small amount of power and require little area. The substantial security, the performance of those devices as well as the minimum storage space and computational capability are the reason, we implement Light Weight Cryptography algorithms. Those algorithms have certain features. They process at low power consumption, low communication cost, low area, low energy and small processing time [1]. For security in Smartphones, tablets, medical implants and wireless sensor networks, that lack resources and power consumption a lightweight stream cipher is proposed in [1], [3].

4. Background on Related Work

Some important projects on embedded systems to provide security on applications that were discussed in the previous chapter are presented in brief in this one. One processor system with a general-purpose processor with a cryptographic processor that performs cryptographic operations and enforces security on critical parameters should prevent exposure of critical security parameters outside the cryptographic processor and implements a limited Scripting engine to provide highly efficient security [52].

A hardware-based Montgomery multiplier, and pairing software is proposed in [3] but lack in some sections. a) Session keys must be embedded in each node at initial implementation, b) Use of a single session key at multiple nodes requires synchronization; and c) Digital signatures are not possible, since all nodes share the session key [3]. It also shows the increase of power and energy consumption. The design uses optimizations for Miller loop, final exponentiation, and Elliptic Curve Cryptography (ECC) operations. It was designed in VHDL and runs in an ARM Cortex-A9 processor [3].

Demands for security in Embedded Systems is increased every day. In [2] the authors propose cost-efficient hardware that can compute the public-key cryptography with the use of a coprocessor that supports both RSA (range from 256 bit to 2048 bit) and Elliptic Curve Cryptography (ECC). They also propose a small direct memory access (DMA) to remove system-level performance bottlenecks and transfer data between coprocessor and external RAM in order to make full use of the coprocessor [2].

A hardware-software co-design of RSA is analyzed in [35] where hardware accelerators are providing the highest performance but fail in flexibility and adaptability to changing algorithms, parameters, and key sizes. The authors estimate that they combine software and hardware that offer high performance and the advantage of low-power and low energy consumption [35]. Test of this design was performed on the Xilinx Zynq- 7000 SoC platform, which integrates a dual-core ARM CortexA9 processing system and the RELIC library (Efficient Library for Cryptography) was used. They show the speed up of the co-design vs the pure software implementation of the RSA algorithm that run on the same platform [35]. An RSA implementation for system on programmable chip (SoPC) can be investigated in [37]. The use of the RSA processor is believed to be at low cost, more flexible and high performance when implemented in a FPGA platform [37]. It is a hardware/software integration of RSA in verilog HDL language and was tested on an Altera Cyclone II FPGA [37]

One cryptographic processor for security in embedded systems is proposed in [4] named CryptoAeg. In difference from general SoC-based solutions, CryptoAeg has its own cryptographic instructions to accelerate cryptographic processing by eliminating the need of coprocessors and minimize the cost of the system for the use of RSA or ECC [4]. The design relies on the ALU architecture without including a multiplier in order to reduce the hardware cost and complexity of software program [4]. It provides security for portable devices and wireless communications [4].

Public key encryption algorithms are very cost efficient when used in wireless sensor nodes, yet if necessary, the best to use is Elliptic Curve Cryptography (ECC) for short keys [24]. Public-key algorithms are more computation-intensive than other types of crypto algorithms like symmetric-key algorithms and hash functions, but are an importance when talking about digital signature and authentication [24]. The performance of the ECC is tested on DSP and it shows efficient results, although it does not run on a general-purpose processor but hardware features of DSP to accelerate the ECC operations are used for this reason [24]. Same in [25], a three side-channel protected hardware/software co-design for a small but particularly fast pairing-based cryptography in a stand-alone microprocessor seems to be another good option when referring to embedded applications because of the low chip area needed. With an assembly optimized software implementation, the low area requirement and the high runtime of pairing-based cryptography the option is viable in interactive embedded applications when involving wireless sensor networks [25].

Elliptic Curve Cryptography for MSP430-Based Wireless Sensor Nodes is a new energy save architecture, a dedicated hardware module for area and speed-optimized software solution [32]. It uses 4kGE of dedicated chip area and consumes less energy but the efficiency of the algorithm in speed is the same as before [32]. One hardware/software Co-design of Elliptic Curve Cryptography is provided in [34] for the 8051 Microcontroller. It uses a minimalist hardware accelerator for ECC and a dedicated interface with direct memory access for better performance and reduction of hardware cost in comparison with previous work on similar 8-bit platforms [34].

Nowadays, an electronic computational unit is embedding more data and power management in connected, autonomous and electric vehicles [5]. The authors in [5] refer to a Reconfigurable hardware technology combined with static multicore processors and memory into a SoC for embedded cryptosystems. Their work focuses on the HW/SW co-design of a secure automotive computation unit which is composed of a full post-quantum cryptosystem implemented in programmable logic to prevent intruder’s decryption using quantum computers

[5]. To achieve this, they implemented the McEliece algorithm, a true random number generator, the advanced encryption standard and a secure hash algorithm. They used the Zynq UltraScale+ MPSoC ZCU102 evaluation board to test their design.

In [6] the authors propose a security coprocessor with the general-purpose processor in an AMBA bus and an AXI interface in a MPSoC. SM2 asymmetric encryption algorithm and SM3 hash function is used to ensure the security in the system [6]. The accelerator in the security coprocessor mainly executes computational functions like Modular multiplication and modular addition. This SoC could be used in a power distribution network as a smart control terminal or a smart metering and power quality analysis instrument [6].

The authors in [10] propose a cryptographic accelerator to multiple cryptographic tasks for Internet Protocol security (IPsec) by using a Dynamically Reconfigurable Processor (DRP) from NEC electronics in a SoC with the embedded processor. The accelerator provides high-throughput cryptographies, cost efficiency, and high flexibility to embedded systems and presents a practical solution to optimize cost, performance, and power consumption [10]. With the use of Virtual Hardware and Run-Time Configuration based on Double Buffer the simulation results show that co-processing system eliminates a bottleneck of the software execution and achieves performance improvement [10].

As technology rises the authors in [13] present a 65 nm Fulmine secure data analytics System-on-Chip for IoT end-nodes with Convolutional Neural Networks and computer vision. This SoC provides full programmability, low-effort data exchange between processing engines, high speed, and low energy. By combining cores and accelerators within a single tightly-coupled cluster this SoC improves time and energy in a simple software solution with flexibility, high security and sensible budget [13].

A great idea is provided in [15] where the authors make use of a dedicated DMA controller, which encrypts and decrypts data in every transaction according to the function required. The Advanced Encryption Data is used to transform data in each DMA request, that prevents transmitted data to be hacked by an unauthorized user [15]. The idea was developed for the IPhone of Apple Computers but can work for every embedded device. Another great idea presents in [16] where each I/O controller has its own identifier and every identifier its own encryption key. Whenever a DMA transaction is to be performed, a computing device called the cryptographic engine protects the data according to the identifier of the I/O controller [16]. The cryptographic engine has an identifier table with the cryptographic keys of every I/O controller [16]. According to the device id, the cryptographic engine encrypts with a different key every time a DMA request is to be made [16].

Same idea for DMA security developed in [19] with the use of an encryption/decryption unit in the DMA path. For historical reasons the idea that presented in [20] explains the scenario of using a security module for encrypting and decrypting data whenever a DMA request is made [20]. Every encryption or decryption is established through the DMA controller without disturbing the main processor [20]. This scenario was proposed for personal computing but applies still in embedded systems applications. In [22] the authors develop a high-performance DMA design with four channels able to transfer 1.6 megabytes of data every second. This DMA controller uses the advanced microcontroller bus architecture (AMBA) and was implemented in Verilog HDL [22].

One implementation of all three AES algorithms (AES-128, AES-192 and AES-256) can be investigated in [23] using cryptographic accelerator with both ECB and CBC mode as well as SHA-1 and SHA-256 hash algorithms combining the speed of hardware with the flexibility of software. The Sboxes in this implementation are in RAM blocks in order to increase the throughput [23]. The hardware cryptographic accelerator using FPGA technology and the client application was tested in a linux platform on a PC using PCI Express interface [23]. In order to achieve higher computation speed, flexibility and implementation scalability parallelization mechanisms were used for all encryption and hash blocks [23]. A kernel driver was also developed in order to connect the hardware unit to the PC and acquire direct access operations to the hardware resources required by the multitasking environment [23].

One Hardware-Software co-design of Cellular Automata Cryptosystem (CAC) shows experimental results better than DES and comparable to AES [18]. CAC is supposed to be fast in execution because of its small code size, designed for embedded systems with an acceptable level of security [18]. CAC is written in verilog and was acceptable by the time invented because of its simplicity (only four levels of transforms) and the level of security that provides, but obsolete nowadays.

According to the authors of [31], a hardware acceleration of AES Cryptographic Algorithm can be developed for IPSec, to provide secure data at the IP layer but the drawback is that itneeds a lot of computational power. It uses a hardware acceleration of AES ECB and the goal was to secure speed and energy efficiency [31]. They tested it on Xilinx Virtex-6 ML605 on 250 MHz and they achieved 391.25 Mbps throughput [31].

An advanced bus architecture for embedded systems that use AES encryption to improve performance and possibilities is provided in [8], called CDBUS. The benefits using the CDBUS include a) low cost and low pawer control bus, b) dual bus structure, c) high – throughput data bus, d) high - efficient DMA with dynamic arbitration, e) high – performance

AES transfer mode [8]. A testbed provided in [9] shows results for execution of various sha3 algorithms. Those results include a) hardware execution time, b) software execution time, c) HW/SW speed up and d) maximum clock frequency. Results were obtained using vivado design Suite. In [11] is an example of a multi FPGA SoC with 24 microblade performing parallel and pipelined signal processing applications in embedded system to achieve performance and [12] shows an improved DMA controller to boost high speed data transfer in MPU based SOC.

One cryptography co-processor in NoC systems can offer complete functionality with just an integrated DMA, embedded key registers, command priority queues, and AES counter mode of operation (CTR) [36]. The general-purpose processor (GPP) only requests encryption and decryption operations from the co-processor. The scenario proposed is high-performance pipelined AES core with counter mode of operation (CTR) with an integrated DMA module to take the load off the GPP and perform all the cryptographic tasks and data transfers in a scalable NoC design [36]. They used an intergraded AES core from Opencores with key register file in our tile to store keys and the priority command queue [36]. The design was only tested in modelsim and it was estimated to be 230 times faster than the pure software implementation of the same algorithm [36].

Two approaches are presented in [38] to implement the AES encryption algorithm in a multi-processor SoC (MPSoC). The first one is composed of a Network Interface (NI), a Controller, AES and internal memory and for the second, a Network Interface, a plasma processor, AES, Direct Memory Access (DMA) and internal memory [38]. The cost to design a pipelined AES algorithm in FPGA is very low and it provides resource utilization, high speed and high throughput [38]. They developed this design in VHDL with the use of the AXIM platform and the customized AES based crypto module and was tested with modelsim and Virtex6 ML605 from Xilinx [38]. In the first approach the design uses less hardware resources and achieves better performance than the second approach but in both cases the latency has reduced significantly [38].

Constant growth of data transfer requirements in modern embedded systems made the need to implement the AES256 and TDES as hardware IP cores on FPGA platform with the use of AXI interface [39]. This way the performance of data encryption/decryption is approximately 13 to 416 times faster compared to the pure software implementation of the algorithms and in addition takes the offload of the main processor [39]. The results are comparable to modern Intel processors with specific instruction set [39]. The IP cores are modeled in VHDL and tested on a zedboard.

The Maestro architecture described in [43] is a hardware/software co-design with two components, one processor for system initialization and control and the hardware AES core for high performance AES encryption/decryption. The design reaches a very high throughput, through a tightly coupled encryption and round key generation units in encryption unit and ahead of time round key generation in decryption unit [43]. The ten-stage pipelined architecture was considered for the AES engine and the authors believe that it can encrypt or decrypt one block of data in one clock cycle [43]. It is also believed to be cost efficient considering the high throughput. The Altera DE2-115 development and educational FPGA board was used to test this design which includes the Nios II core, Avalon interconnects, SRAM, SDRAM and on chip memory [43].

A secure communication protocol is described in [46] in which cryptographic co-processors are used and the cryptographic key is computed according to a password that was set between the users. The hardware/software co-design uses AES-256 block cipher, is used in real time applications and is supposed to be cost-efficient, high power and high secure hardware structure [46]. It provides high security, portability and speed at low costs [46]. The design was tested in Linux OS on a ZYBO combining the ARM CPU and the FPGA [46].

There are three ways to connect a security module to the embedded processor, a) in the processors Datapath, b) through the internal register file of the processor and c) access through the peripheral bus as a peripheral [47]. An AES-128 security module is implemented on [47] as an IP-Core with a true random number generator (TRNG) for keys and is tested in different platforms [47].

One scenario that a Co-processor can be used in some area of application such as random number generation or hash generation and error correction is applied in [21]. The Co-processor performs forward error correction using BCH and Reed Muller algorithms IP Blocks and SHA-1 IP Block for hash generation in embedded systems [21]. The CP is modeled using Verilog HDL and tested with Altera-Acex FPGA [21].

The main goal when referring to security in essence is a vault manager to protect the firmware if an embedded device from unauthorized access [27]. The authors in [27] propose a hardware vault such as this which includes of a shadow RAM or shadow Cache, flash memory, and a vault manager. The architecture proposed offers good security with a small performance penalty over OS applications [27]. The key points of this architecture include a) the vault is external to the processor, b) assures instruction integrity and c) provides trust on firmware upgrades [27].

In [14] the authors implement a hypervisor named BitVisor with minimized code and a parapass-through driver for ensuring storage encryption in ATA input/output devices. It is designed for virtual machine monitors and mainly for desktop operating systems [14]. Similar in [17] a hardware encryption module in the processor ' s memory access path is used by the processor to secure information. Same as before used mostly for hypervisor and virtual machines. In this work the security module is an application specific integrated circuit (ASIC), a general-purpose processor on field programmable gate array (FPGA), designed and configured to perform security operations for the processing system [17]. Another bare-metal hypervisor running on virtual machine OS is the Silvermont microarchitecture for Intel x86 processors, running Windows is proposed in [33] but it is mainly used in POS machines and industrial embedded devices.

For secure mobile authentication a new architecture is presented to produce cryptographic keys and values for use inside an Enhanced Cryptographic Engine (ECE) [48]. A software Application Programming Interface provide stronger security for commands and data and a software emulator ensures secure communication between multiple computers and mobile devices [48]. A Trusted Execution Environment (TEE) provides secure modification, removal and update on embedded systems used in Automotive or health-care networks [48].

Securing DMA transfers can be also accomplished through virtualization based on hardware/software and provide high security with a formal verification of isolation and availability and a low performance overhead [49]. One software-based parallel cryptographic solution with a parallel memory embedded SIMD matrix processor is proposed in [50] which executes encrypting and decrypting cryptographic algorithms. The architecture is very effective for private information protection that promises low power and small chip area consumption and is intergraded in real time [50].

The contribution in [45] presents one hardware solution for ensuring microcomputer bus systems through a Tree Parity Machine Rekeying Architecture (TPMRA). The TPMRA IP-core is designed for adaptability, low cost terms, variable bus performance requirements, authentication of different bus participants as well as the encryption of chip-to-chip buses [45]. A co-processor module together with an application software encrypts communications between CPU, memory and other hardware through stream cipher techniques, Tree Parity Machine together with a hash algorithm that acts as a key stream generator for authenticated key exchange in AMBA bus system [45]. The design was written in VHDL and tested on a FPGA demonstration system, consisting of several FPGA boards [45].

5. Design and Architecture

In this chapter we describe the architecture and design methodology of the hardware and software infrastructure of the project developed. The important characteristics of an embedded system are speed, size, power, security, integrity, authenticity, reliability, accuracy, adaptability, functionality, cost, power requirements, size and weight. In this project a secure DMA controller is been developed with the additional AES block cipher technique for extended security. DMA controller is developed in high Level Synthesis the execution of which is successful under certain constraints. Different IDs are used for every peripheral on the embedded device with different permissions for every ID for write or read operations.

A successful data transfer over DMA depends on the permissions set by the cryptographic co-processor with the addition of AES encryption on those data for secure transmission. Encryption keys are set by the same cryptographic coprocessor in the beginning of the session using a true random number generator. When a data transfer on the embedded device is in order, according to the ID and the permissions granted, the data will be encrypted and decrypted in the other end by AES IP-Blocks also developed in High Level Synthesis in the purposes of this thesis. The design is explained in detail in the sections following.

5.1 DMA Controller

DMA transfers are used for fast data transfer between the IP core or processor and the system memory or the peripherals. In this particular project the DMA is used to transfer secure encrypted data from the cryptographic module to the peripheral or memory that are destined. The end device then uses the key that is provided by the cryptographic co-processor to decrypt the transferred data. The DMA controller has been developed with the Vivado High Level Synthesis (HLS) tool with the use of C programming language, thus, creating an IP block that is used to transfer secure data though the peripherals.

Every peripheral in this design uses a unique device ID for the purposes of communication with other peripherals. The architecture of our DMA controller is designed to receive certain arguments in order to execute a successful DMA transfer. Those arguments are the read and write position of each memory, the quantity of data to be transferred and the ID number for each device. Some boundaries have to be made considering every device ID, because as explained before every device has different rights and permissions. Those boundaries are inserted from a channel as arguments different than the channel we use for the main arguments and is operated exclusively by the cryptographic co-processor.

The cryptographic co-processor is a Microblaze microprocessor used as a shadow in this design which delivers the cryptographic keys to CPU, memory and peripheral devices and sets the arguments-boundaries for each of those devices according to the device ID. After that, every other main process or data transfer is performed by the main CPU, the ZYNQ processing system that includes the ARM 9 two core microprocessor. A main schematic of this project can be shown below.

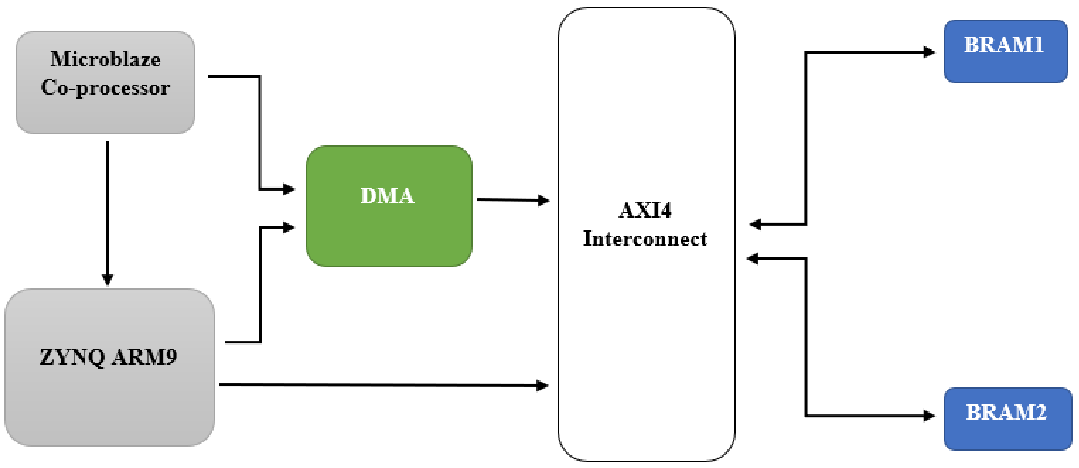
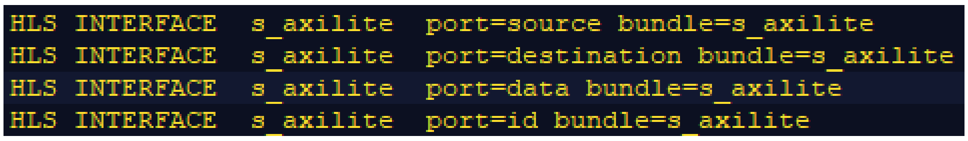


Figure 9: Project Schematic

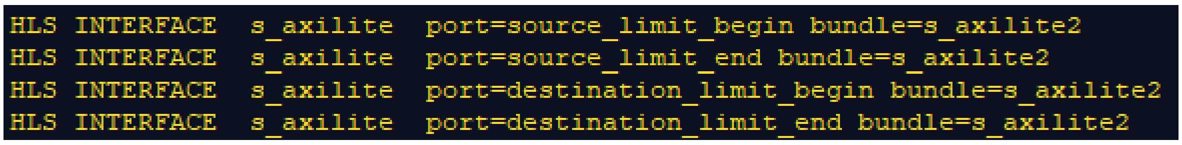
As it can be seen the figure above the AXI-4 environment interface protocol is used for the communication between CPU, memory and peripherals. The DMA IP block has two channels, one for the boundaries provided by the Microblaze microprocessor and the second for the main arguments of the application. Two BRAMs are used in this case scenario in order to test the liability of the system. As the systems boots the application running on the microblaze applies the restrictions on the DMA module and after that, every DMA transfer requested by the applications running on the ARM9 has to apply to those restrictions. The BRAMs are just used as to transfer data from one to the other according on the boundaries given from the microblaze.

The architecture of the design will be explained in more detail, in the sections below. As mentioned Vivado High Level Synthesis is used to build this DMA controller module. As a start the two channels have to be separated, one for the boundaries and one for the main arguments. This is a very sensitive directive as it controls who has rights to use this channel and who doesn't and has been made clear who is supposed to do what. DMA is designed with a master AXI port to associate with the peripheral devices and two AXI slave ports, one for the communication with the microblaze and one for ARM9. The channels can be seen in the pictures bellow.



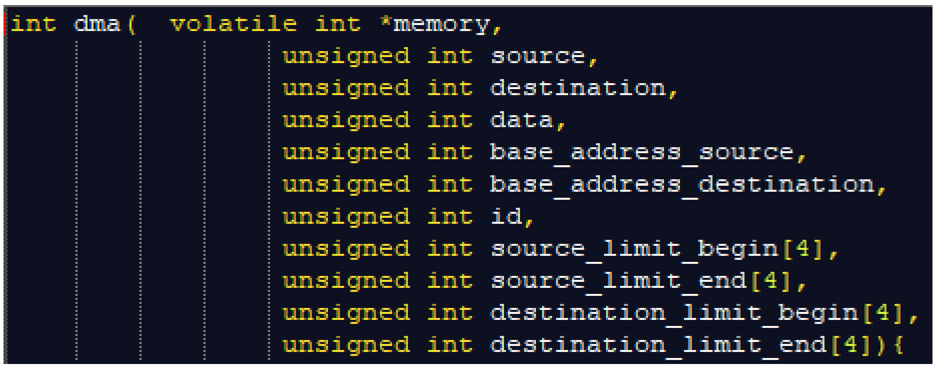
```
HLS INTERFACE  s_axilite  port=source bundle=s_axilite
HLS INTERFACE  s_axilite  port=destination bundle=s_axilite
HLS INTERFACE  s_axilite  port=data bundle=s_axilite
HLS INTERFACE  s_axilite  port=id bundle=s_axilite
```

Figure 10: Main arguments (ARM9)



```
HLS INTERFACE  s_axilite  port=source_limit_begin bundle=s_axilite2
HLS INTERFACE  s_axilite  port=source_limit_end bundle=s_axilite2
HLS INTERFACE  s_axilite  port=destination_limit_begin bundle=s_axilite2
HLS INTERFACE  s_axilite  port=destination_limit_end bundle=s_axilite2
```

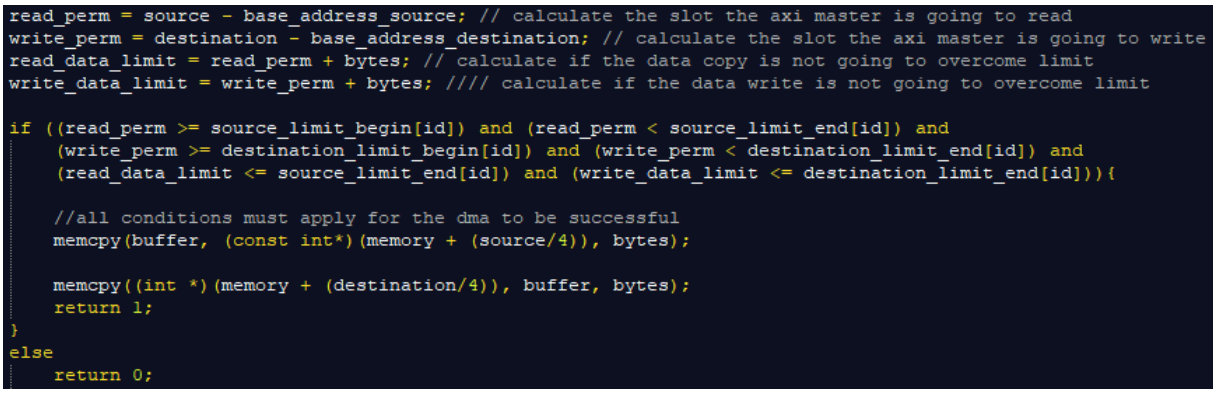
Figure 11: Boundaries Channel (Microblaze)



```
int dma(  volatile int *memory,
          unsigned int source,
          unsigned int destination,
          unsigned int data,
          unsigned int base_address_source,
          unsigned int base_address_destination,
          unsigned int id,
          unsigned int source_limit_begin[4],
          unsigned int source_limit_end[4],
          unsigned int destination_limit_begin[4],
          unsigned int destination_limit_end[4]){
```

Figure 12: Ports of DMA

The functionality of the specific design of DMA depends on the ports shown above. Once the boundaries of the application have been inserted and the main application starts, if a DMA transfer request occurs, the algorithm calculates if the source and destination limits are exceeded. A successful DMA depends whether or not these limits are exceeded. A part of this algorithm written in C Language is shown next.



```
read_perm = source - base_address_source; // calculate the slot the axi master is going to read
write_perm = destination - base_address_destination; // calculate the slot the axi master is going to write
read_data_limit = read_perm + bytes; // calculate if the data copy is not going to overcome limit
write_data_limit = write_perm + bytes; //// calculate if the data write is not going to overcome limit

if ((read_perm >= source_limit_begin[id]) and (read_perm < source_limit_end[id]) and
    (write_perm >= destination_limit_begin[id]) and (write_perm < destination_limit_end[id]) and
    (read_data_limit <= source_limit_end[id]) and (write_data_limit <= destination_limit_end[id])){

    //all conditions must apply for the dma to be successful
    memcpy(buffer, (const int*)(memory + (source/4)), bytes);

    memcpy((int *)(memory + (destination/4)), buffer, bytes);
    return 1;
}
else
    return 0;
```

Figure 13: Calculating the Limits Source and Destination

One benefit arises from this architecture, the use of IDs provides device authentication in the system, so that no other party can use the same channel. The advantage of using priorities for those IDs make it even a stronger security mechanism.

After a successful Synthesis, a Register Transistor Level (RTL) IP Block is exported in order to be used in the general design of this project in Vivado. The security is provided by the hardware, while the memory permissions for each ID are given by one of the applications running over the hardware. In each call for a DMA execution the main application inserts the user's ID, the memory location of the BRAM memory that is going to read, the location of the BRAM memory that is going to write, as well as the quantity of the data to be moved as arguments to the DMA block. If the permissions being set for the certain ID are valid and do not exceed the memory limits for the two BRAM data will be successfully moved. In other case DMA will fail due to denied access (read/write) on one or both BRAMs.

While it is strange to connect two BRAMs in the same AXI master channel and not use two it is perfectly explainable. Direct copy from source memory to destination memory is not permitted so we have to use a buffer in our architecture. Data are firstly copied to the buffer and then from the buffer to the destination, so a second AXI master channel is a waste of space. The AXI interconnect we use is to connect the two BRAMs to one AXI master channel as well as to prevent timing issues.

DMA controller has been designed in a way that every data transfer between two BRAMs can be accomplished only under certain conditions and that's the reason these restrictions cannot be exceeded. The DMA restrictions are accomplished straight on the hardware design of DMA block that is responsible to perform data transfers and without the initialization of these restrictions every call for DMA transfer will be a failure.

5.2 AES IP-Blocks

Embedded systems have sufficient resources such as memory, power and size, thus, they are not able to provide most of the existing cryptographic algorithmic codes. In this section **Lightweight Cryptography** is developed for those devices. Secure processing in this design is accomplished with an embedded cryptographic unit. AES security cryptographic algorithm was chosen for this particular project for its simplicity and the level of security that provides. Two modules created to encrypt and decrypt data, one for encryption and one for decryption algorithm. The reason the two modules are developed separate is to avoid using more chip area than is necessary. Data are encrypted in the one end and decrypted in the other end by different encryption/decryption modules. In that way a higher level of security and integrity is achieved.

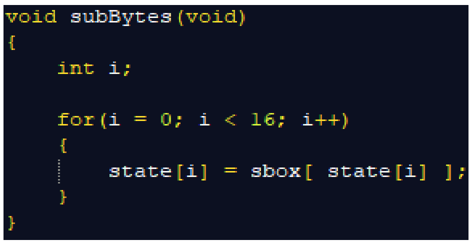
5.2.1 AES Encryption Module

A straight-forward implementation of the AES cryptographic algorithm is used on this project developed at Computer Science Department from University of Santa Barbara (http://cs.ucsb.edu/~koc/cs178/projects/JT/avr_aes.html). It was selected because it is a simple implementation of the AES and the authors of it have given permission to use, copy, modify and distribute of this code for any purpose. In order to use this code as a hardware module, changes have to be made in the code. All details and changes are been explained as follows.

The Advanced Encryption Standard, as explained in Chapter 2, has four steps of operations. All steps are executed in the order they are described except the last transformation where the "“*MixColumns*” is not executed. These four steps are analyzed in C-code.

5.2.1.1 SubBytes

Replacement of the byte in every given byte of input from the lookup table (sbox).

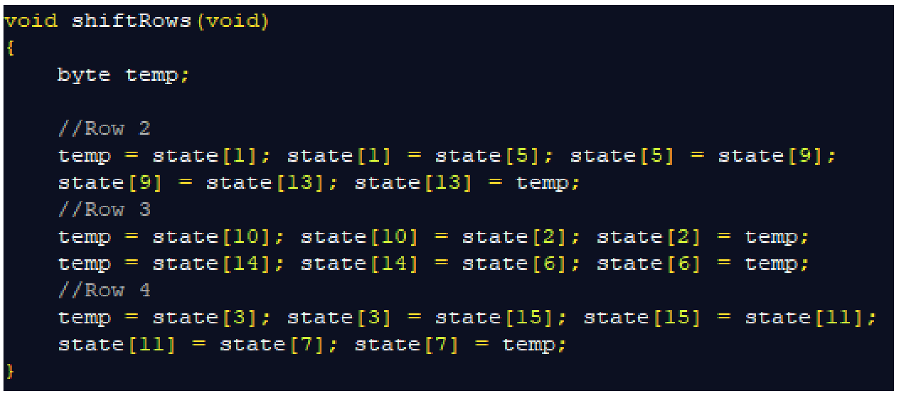


```
void subBytes(void)
{
    int i;

    for(i = 0; i < 16; i++)
    {
        state[i] = sbox[ state[i] ];
    }
}
```

Figure 14: SubBytes in C-code

5.2.1.2 ShiftRows

The rows of the table shift to the left.

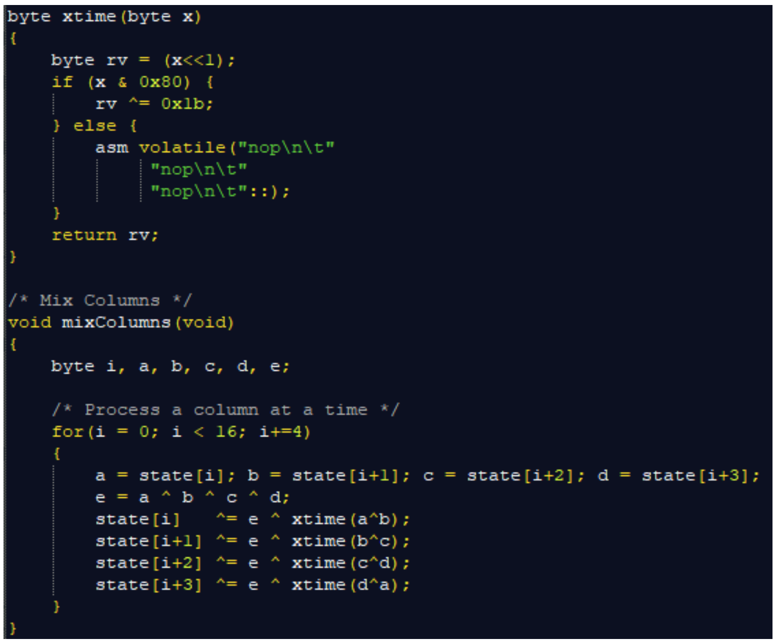


```
void shiftRows(void)
{
    byte temp;

    //Row 2
    temp = state[1]; state[1] = state[5]; state[5] = state[9];
    state[9] = state[13]; state[13] = temp;
    //Row 3
    temp = state[10]; state[10] = state[2]; state[2] = temp;
    temp = state[14]; state[14] = state[6]; state[6] = temp;
    //Row 4
    temp = state[3]; state[3] = state[15]; state[15] = state[11];
    state[11] = state[7]; state[7] = temp;
}
```

Figure 15: ShiftRows in C-code

5.2.1.3 MixColums

Each byte of a column is replaced of all the bytes in the same column.

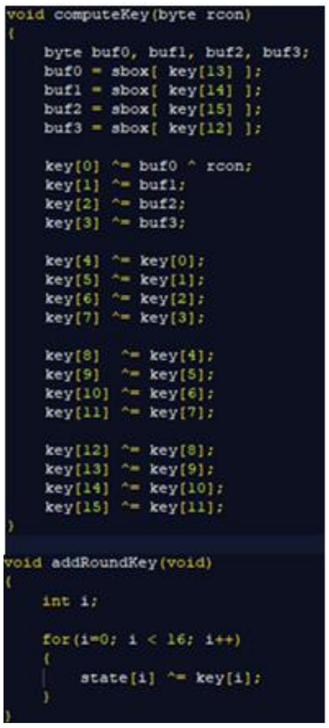


```c
byte xtime(byte x)
{
    byte rv = (x<<1);
    if (x & 0x80) {
        rv ^= 0x1b;
    } else {
        asm volatile("nop\n\t"
                     "nop\n\t"
                     "nop\n\t"::);
    }
    return rv;
}

/* Mix Columns */
void mixColumns(void)
{
    byte i, a, b, c, d, e;

    /* Process a column at a time */
    for(i = 0; i < 16; i+=4)
    {
        a = state[i]; b = state[i+1]; c = state[i+2]; d = state[i+3];
        e = a ^ b ^ c ^ d;
        state[i]   ^= e ^ xtime(a^b);
        state[i+1] ^= e ^ xtime(b^c);
        state[i+2] ^= e ^ xtime(c^d);
        state[i+3] ^= e ^ xtime(d^a);
    }
}
```

Figure 16: MixColums in C-code

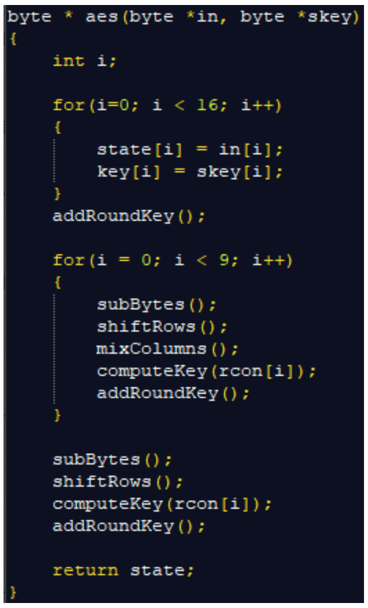
5.2.1.4 AddRoundKey

The round key is computed in every round, in which every Byte is replaced with another using the lookup table and the XOR logical operation.



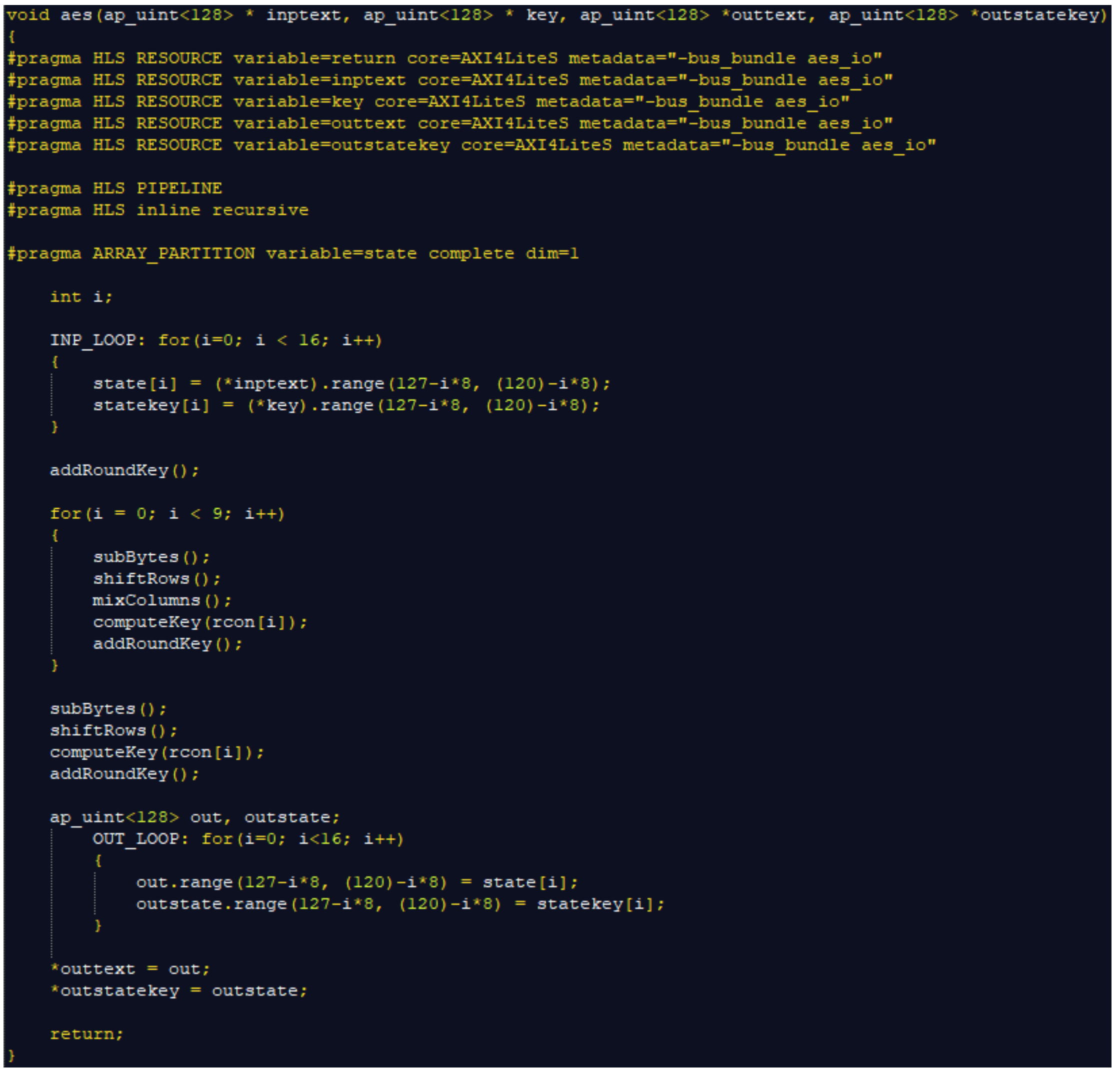
```c
void computeKey(byte rcon)
{
    byte buf0, buf1, buf2, buf3;
    buf0 = sbox[ key[13] ];
    buf1 = sbox[ key[14] ];
    buf2 = sbox[ key[15] ];
    buf3 = sbox[ key[12] ];

    key[0] ^= buf0 ^ rcon;
    key[1] ^= buf1;
    key[2] ^= buf2;
    key[3] ^= buf3;

    key[4] ^= key[0];
    key[5] ^= key[1];
    key[6] ^= key[2];
    key[7] ^= key[3];

    key[8]  ^= key[4];
    key[9]  ^= key[5];
    key[10] ^= key[6];
    key[11] ^= key[7];

    key[12] ^= key[8];
    key[13] ^= key[9];
    key[14] ^= key[10];
    key[15] ^= key[11];
}
void addRoundKey(void)
{
    int i;

    for(i=0; i < 16; i++)
    {
        state[i] ^= key[i];
    }
}
```

Figure 17: AddRoundKey in C-code



```
byte * aes(byte *in, byte *skey)
{
    int i;

    for(i=0; i < 16; i++)
    {
        state[i] = in[i];
        key[i] = skey[i];
    }
    addRoundKey();

    for(i = 0; i < 9; i++)
    {
        subBytes();
        shiftRows();
        mixColumns();
        computeKey(rcon[i]);
        addRoundKey();
    }

    subBytes();
    shiftRows();
    computeKey(rcon[i]);
    addRoundKey();

    return state;
}
```

Figure 18: Main function to produce the ciphertext



```
void aes(ap_uint<128> * inptext, ap_uint<128> * key, ap_uint<128> *outtext, ap_uint<128> *outstatekey)
{
#pragma HLS RESOURCE variable=return core=AXI4LiteS metadata="-bus_bundle aes_io"
#pragma HLS RESOURCE variable=inptext core=AXI4LiteS metadata="-bus_bundle aes_io"
#pragma HLS RESOURCE variable=key core=AXI4LiteS metadata="-bus_bundle aes_io"
#pragma HLS RESOURCE variable=outtext core=AXI4LiteS metadata="-bus_bundle aes_io"
#pragma HLS RESOURCE variable=outstatekey core=AXI4LiteS metadata="-bus_bundle aes_io"

#pragma HLS PIPELINE
#pragma HLS inline recursive

#pragma ARRAY_PARTITION variable=state complete dim=1

    int i;

    INP_LOOP: for(i=0; i < 16; i++)
    {
        state[i] = (*inptext).range(127-i*8, (120)-i*8);
        statekey[i] = (*key).range(127-i*8, (120)-i*8);
    }

    addRoundKey();

    for(i = 0; i < 9; i++)
    {
        subBytes();
        shiftRows();
        mixColumns();
        computeKey(rcon[i]);
        addRoundKey();
    }

    subBytes();
    shiftRows();
    computeKey(rcon[i]);
    addRoundKey();

    ap_uint<128> out, outstate;
        OUT_LOOP: for(i=0; i<16; i++)
        {
            out.range(127-i*8, (120)-i*8) = state[i];
            outstate.range(127-i*8, (120)-i*8) = statekey[i];
        }

    *outtext = out;
    *outstatekey = outstate;

    return;
}
```

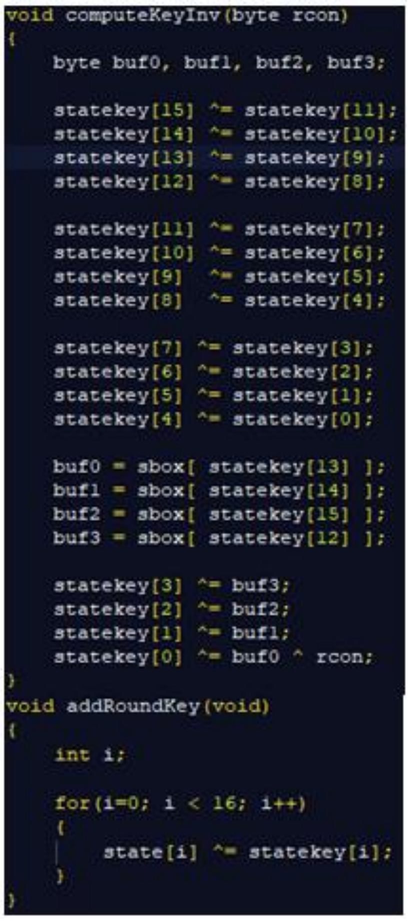
Figure 19: Changes made in order to synthesize a Hardware IP Block

5.2.2 AES Decryption Module

On the AES Decryption Algorithm inversive steps are executed in the reverse mode operation. The C-code for this implementation did not actually exist so it had to be reverse engineered from the Encryption C-code. The inversive steps and the main operation of the AES decryption mode are shown below. Again, all steps are executed in the order they are described except the last transformation where the "“*MixColumnInv*” is not executed.

5.2.2.1 AddRoundKey

The round key is computed in every round in which every Byte is replaced with another using the lookup table and the XOR logical operation.

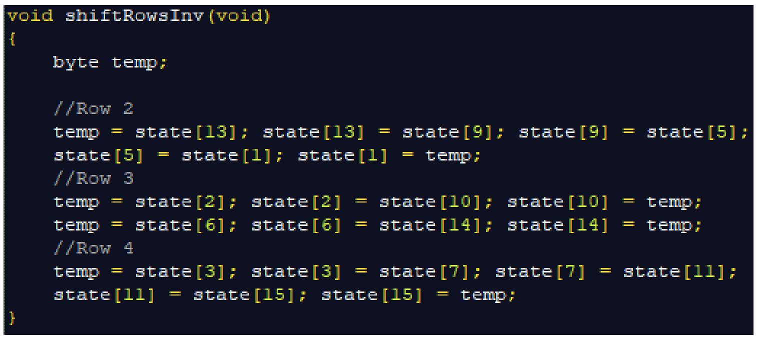


```
void computeKeyInv(byte rcon)
{
    byte buf0, buf1, buf2, buf3;

    statekey[15] ^= statekey[11];
    statekey[14] ^= statekey[10];
    statekey[13] ^= statekey[9];
    statekey[12] ^= statekey[8];

    statekey[11] ^= statekey[7];
    statekey[10] ^= statekey[6];
    statekey[9]  ^= statekey[5];
    statekey[8]  ^= statekey[4];

    statekey[7] ^= statekey[3];
    statekey[6] ^= statekey[2];
    statekey[5] ^= statekey[1];
    statekey[4] ^= statekey[0];

    buf0 = sbox[ statekey[13] ];
    buf1 = sbox[ statekey[14] ];
    buf2 = sbox[ statekey[15] ];
    buf3 = sbox[ statekey[12] ];

    statekey[3] ^= buf3;
    statekey[2] ^= buf2;
    statekey[1] ^= buf1;
    statekey[0] ^= buf0 ^ rcon;
}
void addRoundKey(void)
{
    int i;

    for(i=0; i < 16; i++)
    {
        state[i] ^= statekey[i];
    }
}
```

Figure 20: AddRoundKey in C-code

5.2.2.2 ShiftRowsInv

The rows of the table shift to the right.

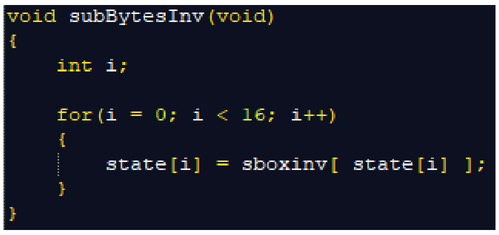


```
void shiftRowsInv(void)
{
    byte temp;

    //Row 2
    temp = state[13]; state[13] = state[9]; state[9] = state[5];
    state[5] = state[1]; state[1] = temp;
    //Row 3
    temp = state[2]; state[2] = state[10]; state[10] = temp;
    temp = state[6]; state[6] = state[14]; state[14] = temp;
    //Row 4
    temp = state[3]; state[3] = state[7]; state[7] = state[11];
    state[11] = state[15]; state[15] = temp;
}
```

Figure 21: ShiftRowsInv in C-code

5.2.2.3 SubBytesInv

Replacement of the byte in every given byte of input from the lookup table (sbox).

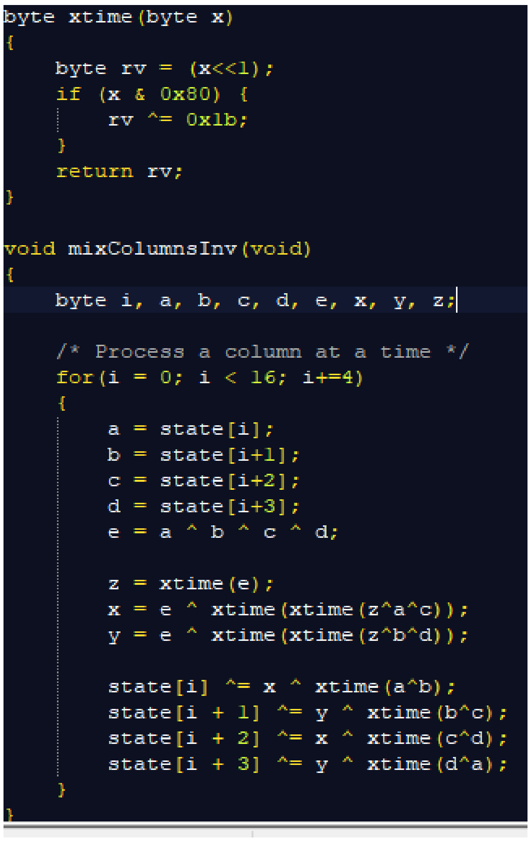


```
void subBytesInv(void)
{
    int i;

    for(i = 0; i < 16; i++)
    {
        state[i] = sboxinv[ state[i] ];
    }
}
```

Figure 22: SubBytesInv in C-code

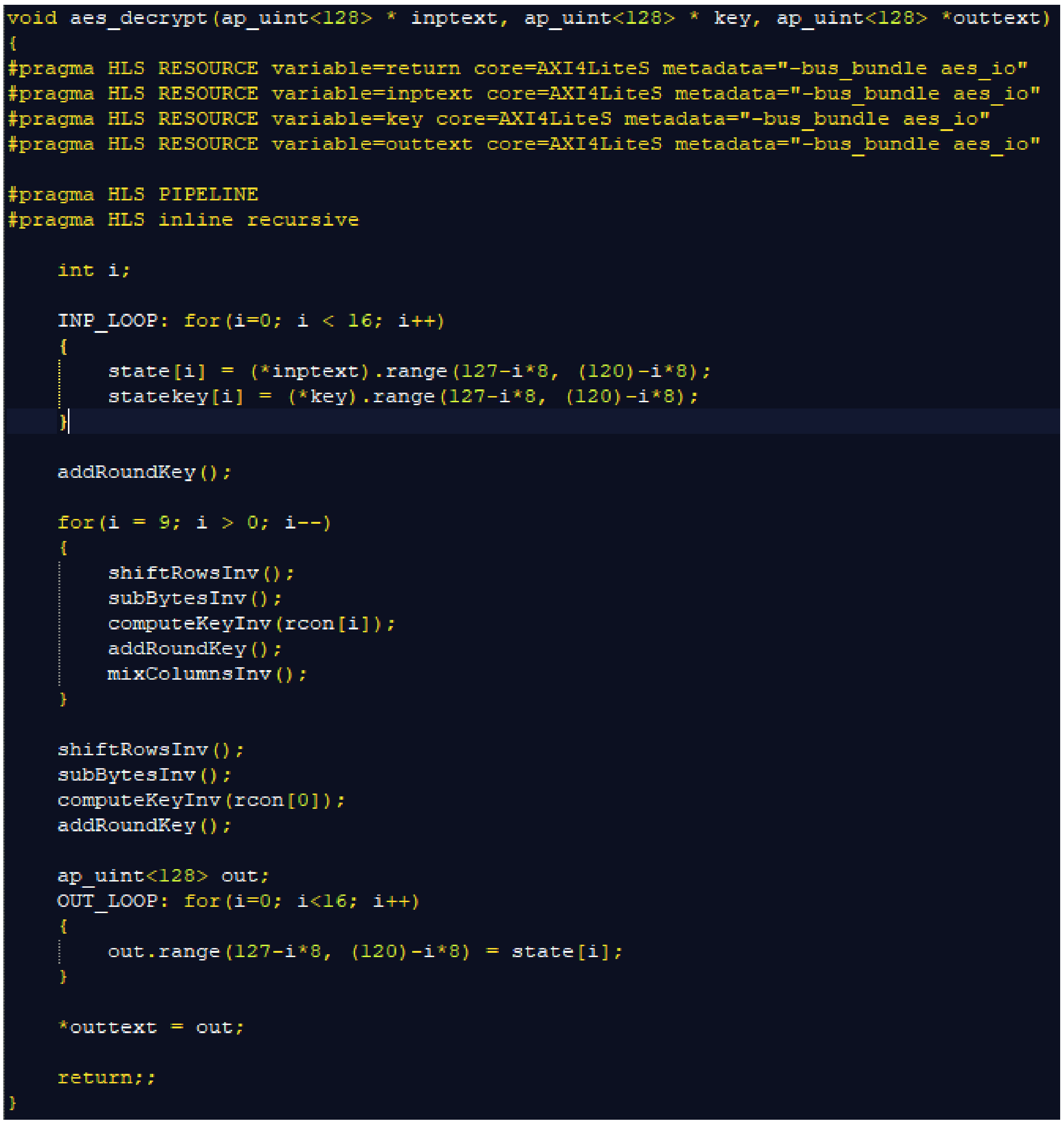
5.2.2.4 MixColumsInv

Each byte of a column is replaced of all the bytes in the same column.



```
byte xtime(byte x)
{
    byte rv = (x<<1);
    if (x & 0x80) {
        rv ^= 0x1b;
    }
    return rv;
}

void mixColumnsInv(void)
{
    byte i, a, b, c, d, e, x, y, z;

    /* Process a column at a time */
    for(i = 0; i < 16; i+=4)
    {
        a = state[i];
        b = state[i+1];
        c = state[i+2];
        d = state[i+3];
        e = a ^ b ^ c ^ d;

        z = xtime(e);
        x = e ^ xtime(xtime(z^a^c));
        y = e ^ xtime(xtime(z^b^d));

        state[i] ^= x ^ xtime(a^b);
        state[i + 1] ^= y ^ xtime(b^c);
        state[i + 2] ^= x ^ xtime(c^d);
        state[i + 3] ^= y ^ xtime(d^a);
    }
}
```

Figure 23: MixColumsInv in C-code



```
void aes_decrypt(ap_uint<128> * inptext, ap_uint<128> * key, ap_uint<128> *outtext)
{
#pragma HLS RESOURCE variable=return core=AXI4LiteS metadata="-bus_bundle aes_io"
#pragma HLS RESOURCE variable=inptext core=AXI4LiteS metadata="-bus_bundle aes_io"
#pragma HLS RESOURCE variable=key core=AXI4LiteS metadata="-bus_bundle aes_io"
#pragma HLS RESOURCE variable=outtext core=AXI4LiteS metadata="-bus_bundle aes_io"

#pragma HLS PIPELINE
#pragma HLS inline recursive

    int i;

    INP_LOOP: for(i=0; i < 16; i++)
    {
        state[i] = (*inptext).range(127-i*8, (120)-i*8);
        statekey[i] = (*key).range(127-i*8, (120)-i*8);
    }

    addRoundKey();

    for(i = 9; i > 0; i--)
    {
        shiftRowsInv();
        subBytesInv();
        computeKeyInv(rcon[i]);
        addRoundKey();
        mixColumnsInv();
    }

    shiftRowsInv();
    subBytesInv();
    computeKeyInv(rcon[0]);
    addRoundKey();

    ap_uint<128> out;
    OUT_LOOP: for(i=0; i<16; i++)
    {
        out.range(127-i*8, (120)-i*8) = state[i];
    }

    *outtext = out;

    return;;
}
```

Figure 24: Main function to produce plaintext synthesizable for a Hardware IP Block

5.3 Cryptographic Co-Processor

The objective of this project is achieving high speed of operation with low cost implementation, which is a difficult subject. In order to meet this requirement a Cryptographic co-processor is used to manage cryptographic keys and initialize DMA transfers in secure mode. A microblaze microprocessor was chosen for this purpose, for its rich instruction set, flexibility in a very low cost for a FPGA. One true random number generator (TRNG) is running in the hidden memory of the microblaze to generate keys for every peripheral and memory in the system in order to use them in DMA transfers.

When the system starts the cryptographic co-processor distributes the cryptographic keys to all devices connected to it. After that the co-processor passes arguments to the DMA module in a private and secure channel about the boundaries that every device in the system has. As it was explained, not everyone has the same rights in a secure system. Those rights can only be given by the co-processor according to the specifications that have been set by the system manager. Both the TRNG and the permissions settings are running as software applications in the hidden memory of the cryptographic co-processor, which is the only one that has access to those apps in the system. No other part in the system has access to this part of the system making it reliable and safe for every application.

Once the cryptographic keys have been distributed and the boundaries of the system have been set, any application running on the main CPU can ask for a DMA transfer. If the arguments of the transfer apply to the options set the DMA will transfer cryptographic data form one end to another. In any other situation the transfer request will fail. If the transfer is successful the encrypted data can be decrypted at the other end by the key that was distributed and by that end only since no other end has the key for this data if not allowed. Nor the CPU nor any other peripheral in the system has access to this co-processor, but only to request new generation of cryptographic keys or request new DMA restrictions.

The CPU executes DMA transfers in secure mode. When DMA is finished sends an interrupt to the CPU to inform whether the data were secure transmitted or otherwise failed. A DMA transfer moves data encrypted from source to destination only if the communication between the two ends is allowed by the restrictions set, thus, making it a fast and reliable communication mechanism in embedded systems applications.

5.4 True Random Number Generator

A True Random Number Generator (TRNG) is used to generate cryptographic keys for every pair of devices connected in the system. Every pair of devices means a different ID, which means that number of keys depends on how many devices are connected to the system and how many of them have communicational rights between them. The scenario for key generation can be changed and depends on the security the user is trying to achieve. Keys can be generated in the beginning of the session and then every month, week, day or even hour. The algorithm creates 128 bits random keys as a function of time every time it runs and for as many IDs as demanded. By taking in mind the time of the request for the key, XORing that time with the time the previous key was generated gives a range of random numbers in which the generator chooses randomly some of them, thus, creating a key.

The original idea was to synthesize a true random number generator in a hardware module, but since some of its functions are not synthesizable, it was decided to run it as a software function running on the hidden memory of the microblaze microprocessor or on this case the cryptographic co-processor. Only the cryptographic co-processor can execute this function and it is the only one who can distribute those keys to the peripheral devices. The main CPU has not access to this software application nor has any other device, except of requesting new cryptographic keys. By providing these rules, the system becomes very reliable and safe and thus, providing confidentiality, authenticity, and integrity.

5.5 Secure Embedded System

By combining all these technologies and modules we developed, one secure embedded system can be created providing secure access via DMA protection and lightweight cryptography for fast but reliable communication between peripheral devices. A system of such is responsible to deliver encrypted information from source to destination if the permissions allow it. Those permissions depend on the settings that are set by the manager and basically describe which peripheral can communicate with another, or, when referring to memories, the predefined space given. In another point of view and to make it more understandable, those restrictions refer to physical addresses in the system. Every device has some boundaries on with which devices can communicate. For every pair of transmission an ID is created and different restrictions are set for every ID. Every ID has a priority number in a hierarchical list on using the DMA module to transfer data adding another security mechanism. This can be achieved by an interrupt controller in the system, sending requests for DMA transfers and operating on priority basis.

5.5.1 Schematic Design

The schematic design in the image below shows how the members of the system are connected. The system consists of the main CPU ZYNQ ARM9, the Microblaze cryptographic co-processor, one AES Encryption module, one AES Decryption module, the DMA and two BRAMs used for testing. The modules are connected through AXI interconnects which is the communication protocol that is used. All rules set for this system and explained in the sections above, apply in this design. The functionality of the system as well as the security it provides is explained and tested in the sections following.

Every IP block has been explained in the sections above except the BRAMs and the AXI Interconnect. The AXI Interconnect works as a main channel to connect every part in the system that are communicating with the AXI-4 communication protocol for embedded devices. The two BRAMs are used in this project as an example of peripheral devices, in order to test the functionality of the system. The memory size of those BRAMs has been divided in to sections to act as boundaries, in order to create some IDs to test the DMA and cryptographic co-processor. By dividing the BRAMs into sections the co-processor can pass as arguments (restrictions) from which section one ID can read and in which it can write, making it a perfect example as if there were other peripheral devices.

Operating in that example, it really does not make any difference since the arguments passed are device addresses, as is every memory slot in a memory. With these restrictions set, when data are requested from one section to another and if the DMA transfer is valid according the ID, those data will be encrypted by the AES encryption module and decrypted on the destination by the AES decrypt module. Encryption and decryption on the two sides of communication is accomplished by the cryptographic key assigned on this ID (both sides) by the cryptographic co-processor.

Cryptographic keys are created through a software program running in the hidden memory of the microblaze cryptographic co-processor. It is a true random number generator that generates cryptographic keys for every part in the system. The cryptographic co-processor distributes those keys to every device in the system at the start of the session. The keys can change as how often as the user decides. The software application of the user, running on the main CPU, requests new cryptographic keys by sending a request (interrupt) to the cryptographic co-processor. That can happen as often as is required, every week, day, hour etc.

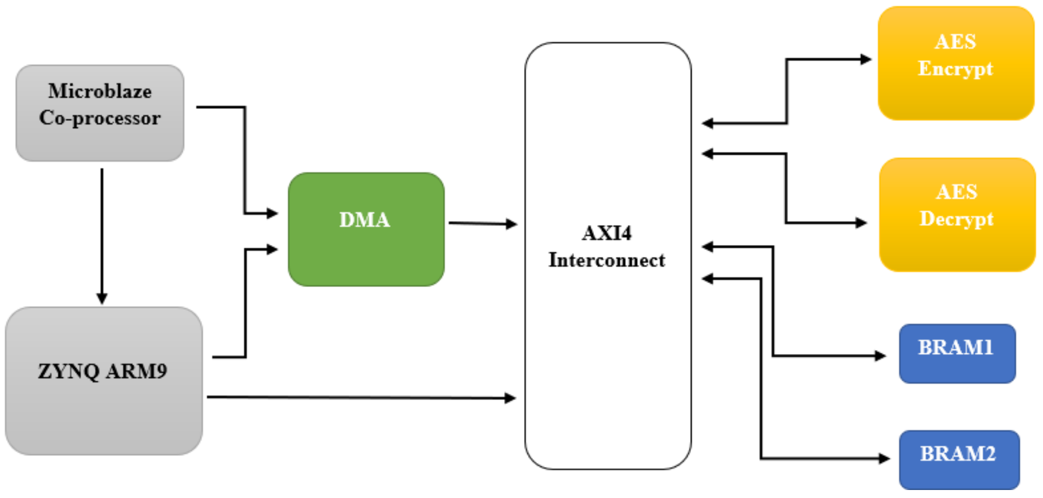


Figure 25: A secure Embedded system (Schematic)

5.5.2 Architectural Implementation

IP blocks are created in Vivado high level synthesis and implemented in Vivado design suite. Vivado design suite is used to integrate the design described in this project. The IP modules DMA, AES Encryption and AES Decryption were developed in C language and synthesized in Vivado high level synthesis. They were exported in IP Blocks and can be implemented in an intergraded hardware design. One base design in Vivado consists of the Zynq ARM 9 processing unit, the DDR memory and standard Input/Output. The IP modules developed in this project are implemented in the base design as well as a microblaze microprocessor to act as a cryptographic co-processor. Two double channel BRAMs are also implemented in the design in order to check the functionality of the system. The architectural design is shown in the image bellow.

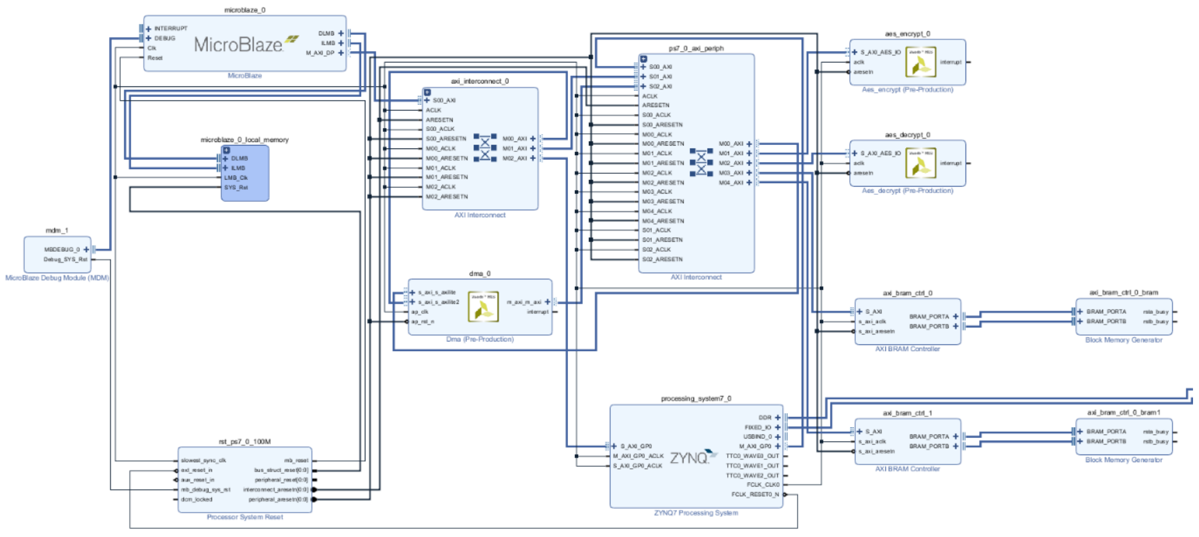


Figure 26: A secure Embedded system (Architecture 1)

Implementation issues and restrictions occur when designing an integrated circuit. Some of them have to do with the connection of the IP blocks, the communication protocol, security protocols, FPGA size, input and output ports, user restrictions, hardware restrictions, software restrictions, etc. The AXI-4 communication protocol was chosen as channel to connect the various IPs. Some constrains rise as who acts as a master or a slave interface. The reason for those constrains is that some of those IPs act both as master and slave. It actually has to do about the role they have in the design. For example, the microblaze cryptographic microcontroller can’t be a slave to anyone since he manages cryptographic keys and in case of a “leak” the results would be catastrophic for the whole system.

Vivado design suite has a number of restrictions when designing. The Zynq ARM 9 obviously has a master interface since he executes all the processes, but in this case, he also has a slave interface for the microblaze microprocessor. The co-processor passes cryptographic keys to the Zynq every time it is requested, so he needs to have master interfaces to distribute those keys. The Microblaze doesn’t have any slave interfaces for the reason mentioned previously. The DMA has two slave interfaces and one master. The master is to read and transfer the data to the peripheral that is requested. One slave is to receive the boundaries form the co-processor and the other for the arguments of the DMA transfer, which are received from the Zynq. The AES blocks have slave interfaces since they only transform data requested for DMA transfers. The BRAMs have slave interfaces by default since they are simple memories. All parties are connected through AXI interconnects, as is required from Vivado design suite.

Users Software applications are executed by the Zynq and are stored in the DDR memory. Boundaries, restrictions and cryptographic keys are applications running in the microblaze co-processor, set by the system manager and are stored in the hidden memory of the microblaze. No other part of the system has access in that memory making in perfectly secure and reliable. The block design shows the wiring between every device in the system and from the connections can be seen that everything claimed in this project applies. If the design is validated from Vivado design suite and the addresses of every cell are correctly registered, the process of synthesis can begin. From the synthesized design, the logic and hierarchy can be examined as well as the utilization efficiency.

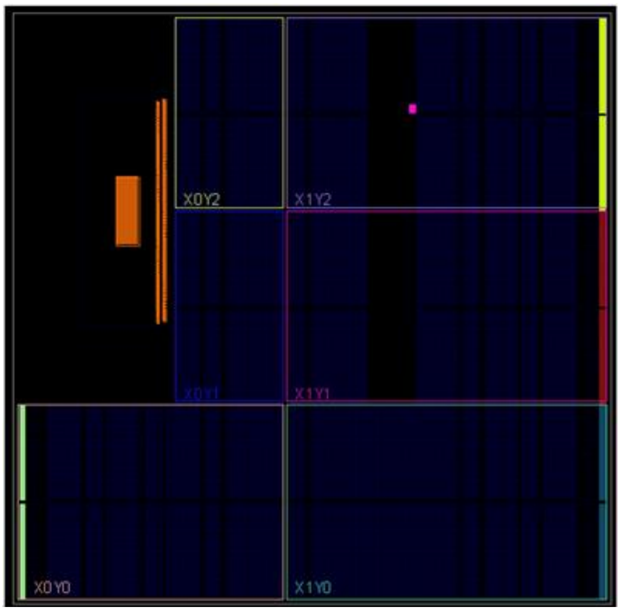
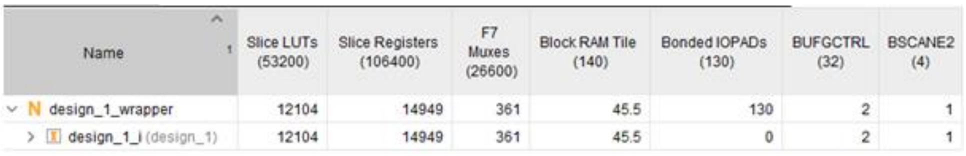


Figure 27: Logic ports from Synthesis



Name	Slice LUTs (53200)	Slice Registers (106400)	F7 Muxes (26600)	Block RAM Tile (140)	Bonded IOPADs (130)	BUFGCTRL (32)	BSCANE2 (4)
design_1_wrapper	12104	14949	361	45.5	130	2	1
design_1_i (design_1)	12104	14949	361	45.5	0	2	1

Figure 28: Utilization efficiency from Synthesis

The successful synthesized design has to be implemented to an FPGA design. That is the next step, the implementation stage. In this stage the design is placed and routed into FPGA resources and the utilization efficiency is examined. If the Implementation stage is successful, the last step is to generate the Bitstream. The bitstream is used to program the FPGA target device, in this case the zedboard.

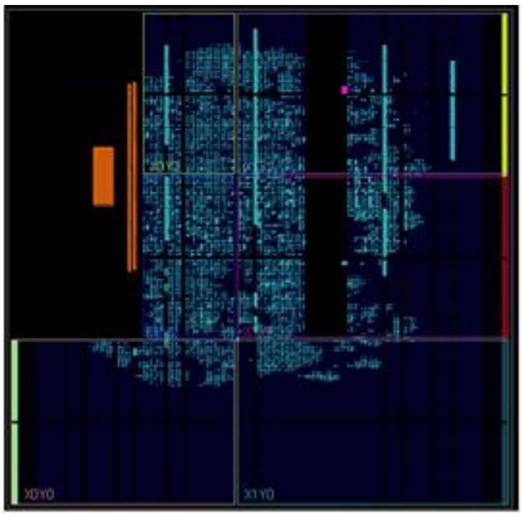
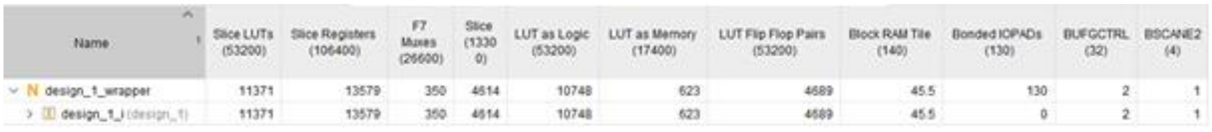


Figure 29: FPGA target From Implementation



Name	Slice LUTs (53200)	Slice Registers (106400)	F7 Muxes (26600)	Slice (13300)	LUT as Logic (53200)	LUT as Memory (17400)	LUT Flip Flop Pairs (53200)	Block RAM Tile (140)	Bonded IOPADs (130)	BUFGCTRL (32)	BSCANE2 (4)
design_1_wrapper	11371	13579	350	4614	10748	623	4689	45.5	130	2	1
design_1_i (design_1)	11371	13579	350	4614	10748	623	4689	45.5	0	2	1

Figure 30: Utilization efficiency from Implementation

If everything and every step is successful the software development kit is used to test the functionality and the performance of the system generated. The zedboard is used as the hardware target to run the applications to be executed. Processes like synthesis, implementation and bitstream generation usually takes a lot of time, so it is essential to try and prevent logical mistakes and errors right at the beginning of the design.

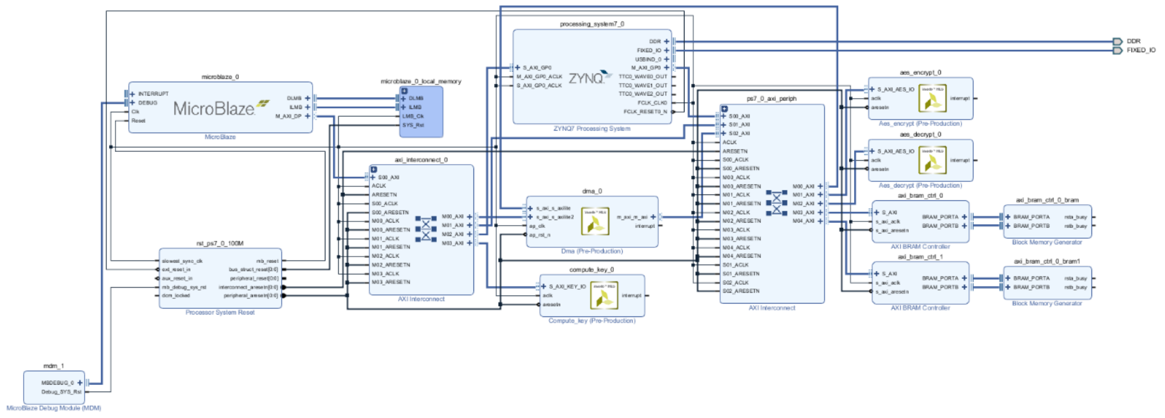


Figure 31: A secure Embedded system (Architecture 2)

6. Evaluation and Performance

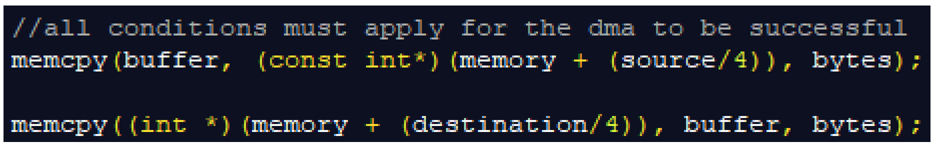
In this chapter, the system designed is evaluated, performance measured and compared to similar existed applications as well as with other architectures developed to provide security in embedded systems. Several of these applications have been described in chapter four. In this chapter this design is tested for its functionality, usability, safety, integrity, reliability, authenticity and speed. Most of the important factors when designing an embedded system apply in this project and all of them are tested thoroughly.

In order to test the design as a whole system, firstly it has to be tested separately, each IP block at a time. The first thing to test is the DMA block that is developed, thereafter the two AES blocks and finally all together as one system. A performance comparison between software and hardware solutions is also an importance in this project. First of all, several scenarios have to be tested in order to gain a global view of each of the blocks that are designed. Test scenarios include multiple blocks for AES encryption/decryption as well as multiple DMAs and one approach that the encryption/decryption modules can operate inside the DMA block.

6.1 Module Evaluation

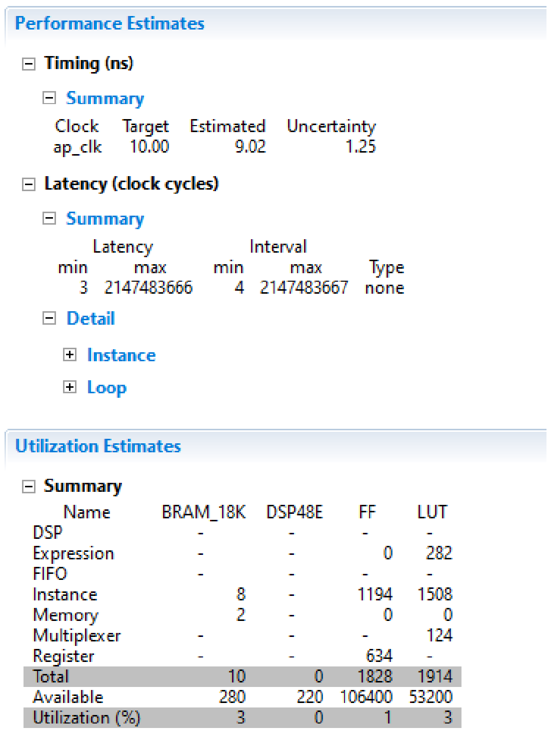
After synthesizing a module, Vivado HLS conducts a synthesis report for the block. When synthesizing one module, the software estimates performance and utilization. The execution time for every block is estimated form the latency multiplied by the period. If we want to increase the performance, we either lower the period or lower the latency. This can be achieved by splitting the operations, but it’s not achievable in every situation. In this case where data are copied from one place to another, splitting operations is not an option. Loop unrolling and pipelining techniques are other ways to improve performance, which cannot be used in the DMA section but have very positive results on the AES blocks, thus, removing bottlenecks and achieve higher throughput.

As far as the DMA concerns, no loops are running in this block and every operation is sequential, thus, no improvement can be made to provide better performance. The execution time depends on the amount of data transferred. The main operation of the DMA core as well as the performance and utilization estimation can be seen in the images below.



```
//all conditions must apply for the dma to be successful
memcpy(buffer, (const int*)(memory + (source/4)), bytes);

memcpy((int *)(memory + (destination/4)), buffer, bytes);
```

Figure 32: Main operation, data copying



Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	9.02	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
3	2147483666	4	2147483667	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	282
FIFO	-	-	-	-
Instance	8	-	1194	1508
Memory	2	-	0	0
Multiplexer	-	-	-	124
Register	-	-	634	-
Total	10	0	1828	1914
Available	280	220	106400	53200
Utilization (%)	3	0	1	3

Figure 33: DMA Synthesis report

From the performance estimation, it takes minimum 3 clock cycles to get going and at least 4 more to receive new data, but that depends on the amount of data waiting to be transferred. The next step is to use the block in Vivado as an independent module. In order to do that, only the ZYNQ processing system is used and two BRAMs as peripheral devices. Each of these BRAMs are divided into four segments in order to create four IDs. Suppose that a 4K BRAM is equally divided in four segments of 1K each. Every BRAM slot is a 32-bit (4 byte) word, thus, providing 256 memory slots for every segment and 1024 in total. Limitations for every segment depend on the specifications of the system manager.

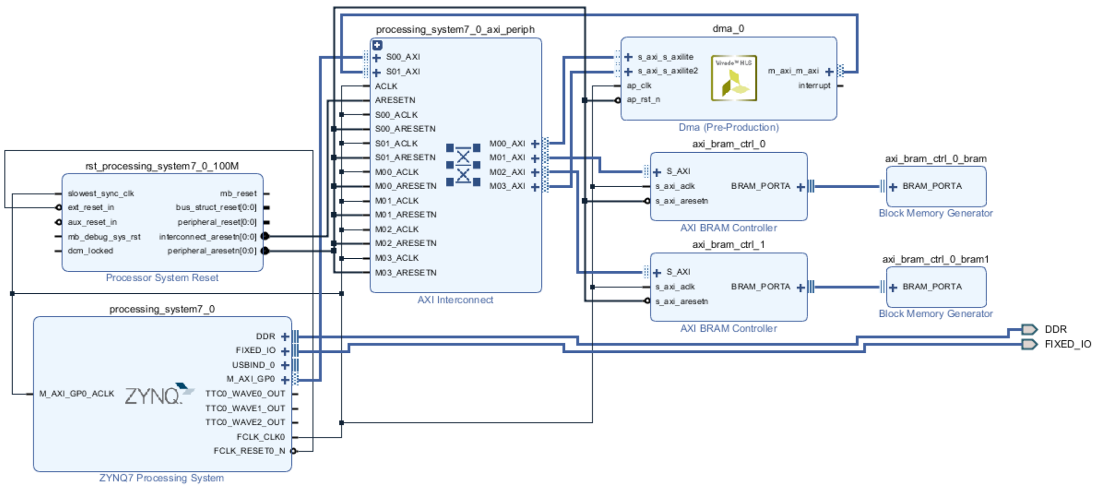
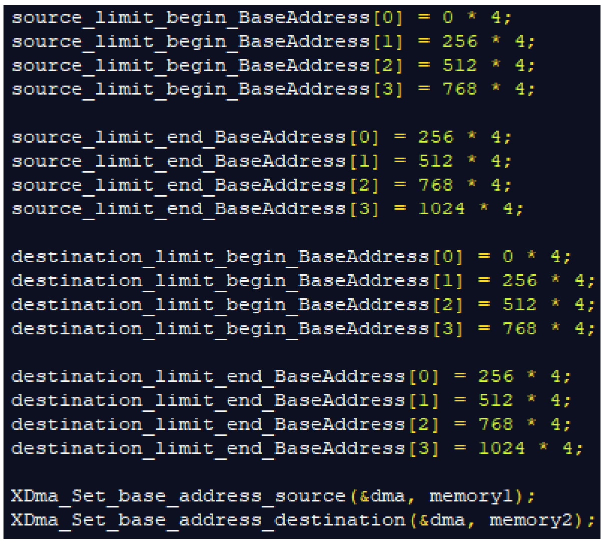


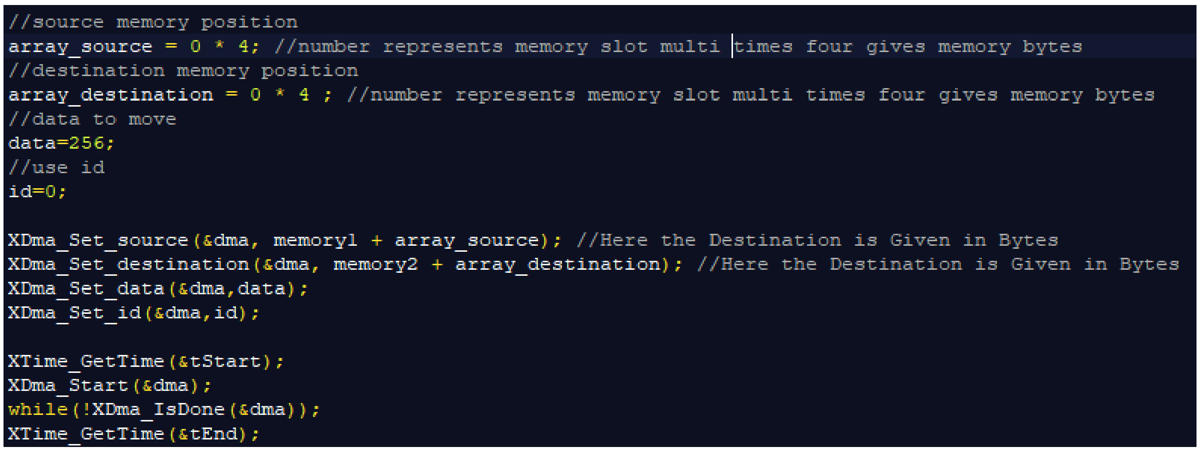
Figure 34: DMA testing

The schematic design of the system for testing is presented in the image above. The design is valid, is synthesized, implemented and a bitstream of it generated. The bitstream is downloaded to the FPGA of the zedboard. At this point the software development kit (SDK) is used to write the software applications, one for the restrictions and one for transfer.



```
source_limit_begin_BaseAddress[0] = 0 * 4;
source_limit_begin_BaseAddress[1] = 256 * 4;
source_limit_begin_BaseAddress[2] = 512 * 4;
source_limit_begin_BaseAddress[3] = 768 * 4;

source_limit_end_BaseAddress[0] = 256 * 4;
source_limit_end_BaseAddress[1] = 512 * 4;
source_limit_end_BaseAddress[2] = 768 * 4;
source_limit_end_BaseAddress[3] = 1024 * 4;

destination_limit_begin_BaseAddress[0] = 0 * 4;
destination_limit_begin_BaseAddress[1] = 256 * 4;
destination_limit_begin_BaseAddress[2] = 512 * 4;
destination_limit_begin_BaseAddress[3] = 768 * 4;

destination_limit_end_BaseAddress[0] = 256 * 4;
destination_limit_end_BaseAddress[1] = 512 * 4;
destination_limit_end_BaseAddress[2] = 768 * 4;
destination_limit_end_BaseAddress[3] = 1024 * 4;

XDma_Set_base_address_source(&dma, memory1);
XDma_Set_base_address_destination(&dma, memory2);
```

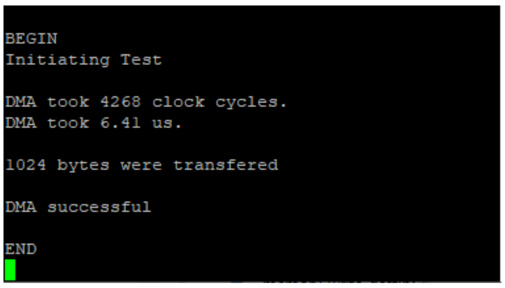
Figure 35: Setting the limits



```
//source memory position
array_source = 0 * 4; //number represents memory slot multi times four gives memory bytes
//destination memory position
array_destination = 0 * 4 ; //number represents memory slot multi times four gives memory bytes
//data to move
data=256;
//use id
id=0;

XDma_Set_source(&dma, memory1 + array_source); //Here the Destination is Given in Bytes
XDma_Set_destination(&dma, memory2 + array_destination); //Here the Destination is Given in Bytes
XDma_Set_data(&dma,data);
XDma_Set_id(&dma,id);

XTime_GetTime(&tStart);
XDma_Start(&dma);
while(!XDma_IsDone(&dma));
XTime_GetTime(&tEnd);
```

Figure 36: Testing the DMA

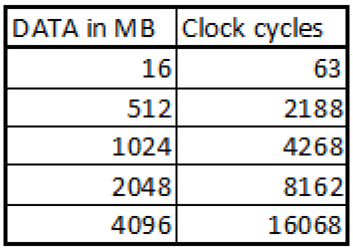
Applications at this time are written, so that the DMA transfer will be successful. The idea is to test the functionality and the time needed to execute a DMA transfer of the amount of 1K of data. For time measurements purpose the ZYNQ global timer is used. It increases by one every two clock cycles and it’s the most accurate timer as it was explained by Xilinx. It took approximately 4268 clock cycles in a bare metal application to transfer the amount of 1KB of data. The Zynq runs at 667 MHz, so it takes 6,41 microseconds to execute a DMA transfer of 1KB of data. It is acceptable since it takes 2000 clock cycles for the zedboard to execute a simple print function in a bare metal application and 250.000 clock cycles for a 1024 time for function that inserts data in two simple 4KB simple BRAMs.



```
BEGIN
Initiating Test

DMA took 4268 clock cycles.
DMA took 6.41 us.

1024 bytes were transfered

DMA successful

END
```

Figure 37: Trasfer of 1KB

Similarly, for different amount of data the following results came up. It is obvious that the execution time for every transfer is proportional to the amount of the data that are to be transferred through the DMA module.



DATA in MB	Clock cycles
16	63
512	2188
1024	4268
2048	8162
4096	16068

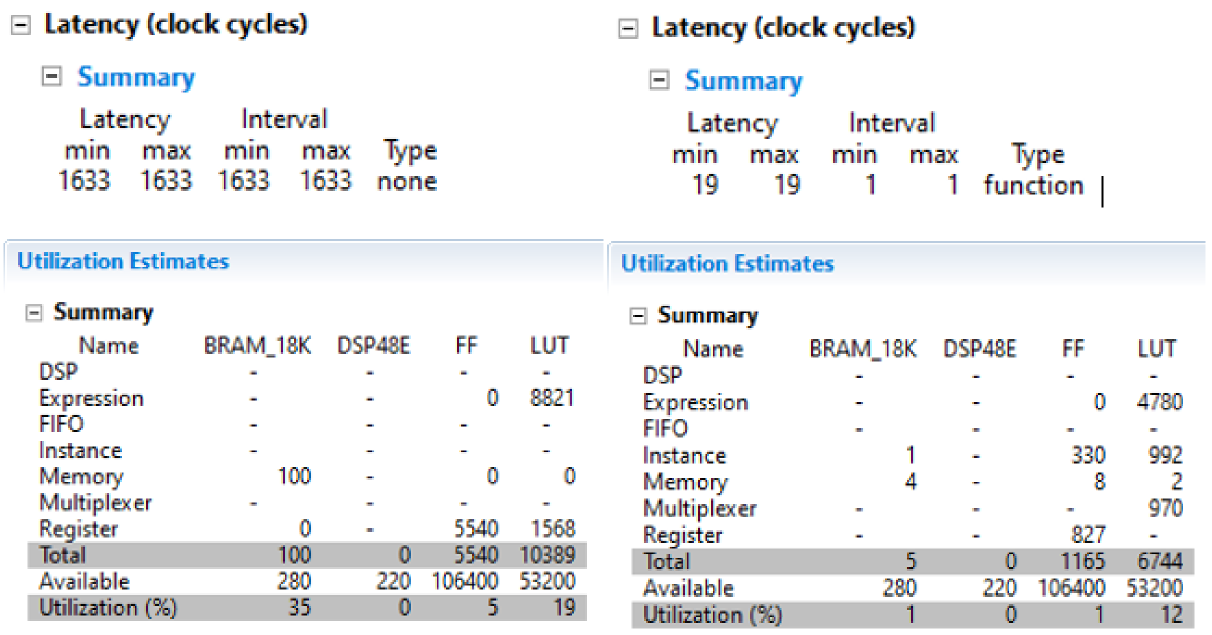
Figure 38: Transfer time for different amount of data

After recognizing that the module operates as it was designed and the execution time is between the acceptable level of tolerance, we procced further to test the next modules developed.

Cryptographic operations are a much more complicated process when it comes to implement them as hardware modules. Since it was successfully achieved to create AES hardware modules in this project, they have to be tested separately and evaluated, as well as performance measured. Before measuring performance, it has to be proved that the encryption and decryption modules are operating as they supposed to. One way to do that is to compare the results from https://www.hanewin.net/encrypt/aes/aes-test.htm and if the results are the same, the modules are successfully operating as the AES algorithm is designed to. AES comes with three variations, AES128, AES192 and AES256 depending on the size of the key. AES256 has better performance results because of the smaller number of iterations in large blocks of data (10 for 128, 12 for 192 and 14 for 256).

Mainly the AES 128bit key is investigated in this project but all results and comparisons apply to every cryptographic algorithm that is implemented in a hardware module. Same things apply with AES 192bit key and AES 256bit key. With two simple changes in the code, every one of the three implementations can be chosen. For simplicity we chose AES 128bit key. It is a block cipher so it processes blocks of data of 128bit (16 Bytes) each or four

32bit words. The two AES blocks are synthesized and exported the RTLs. The synthesis report on both encryption and decryption blocks are the same since the operations are similar for the most part. At this time the Performance estimation and Utilization from the synthesis report can be seen in the image below.



Latency (clock cycles)

Summary

Latency min	Latency max	Interval min	Interval max	Type
1633	1633	1633	1633	none

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	8821
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	100	-	0	0
Multiplexer	-	-	-	-
Register	0	-	5540	1568
Total	100	0	5540	10389
Available	280	220	106400	53200
Utilization (%)	35	0	5	19

Latency (clock cycles)

Summary

Latency min	Latency max	Interval min	Interval max	Type
19	19	1	1	function

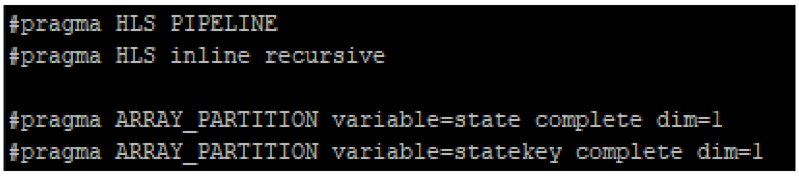
Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	4780
FIFO	-	-	-	-
Instance	1	-	330	992
Memory	4	-	8	2
Multiplexer	-	-	-	970
Register	-	-	827	-
Total	5	0	1165	6744
Available	280	220	106400	53200
Utilization (%)	1	0	1	12

Figure 39: Utilization and Performance estimation of AES encryption/decryption block

Loop unrolling and pipelining techniques where used in these two modules to reduce latency to the minimum and achieve higher data throughput as well as remove bottlenecks. After a few configurations and tests to reduce latency but keep the functionality, the results above where the best that came up. From the performance estimation the synthesizer calculates that it takes 19 clock cycles to output the results but it takes only 1 to take new inputs, so as is has new inputs it can pipeline the operations and generate new results. Next, they have to be implemented to a ZYNQ base design for testing. Again, these two modules have to be tested separately in order to prove the encryption and decryption processes are operating as they supposed to and described by the cryptographic algorithm. The two modules are connected to the base design in a bare metal application.



```
#pragma HLS PIPELINE
#pragma HLS inline recursive

#pragma ARRAY_PARTITION variable=state complete dim=1
#pragma ARRAY_PARTITION variable=statekey complete dim=1
```

Figure 40: Encryption & Decryption modules directives

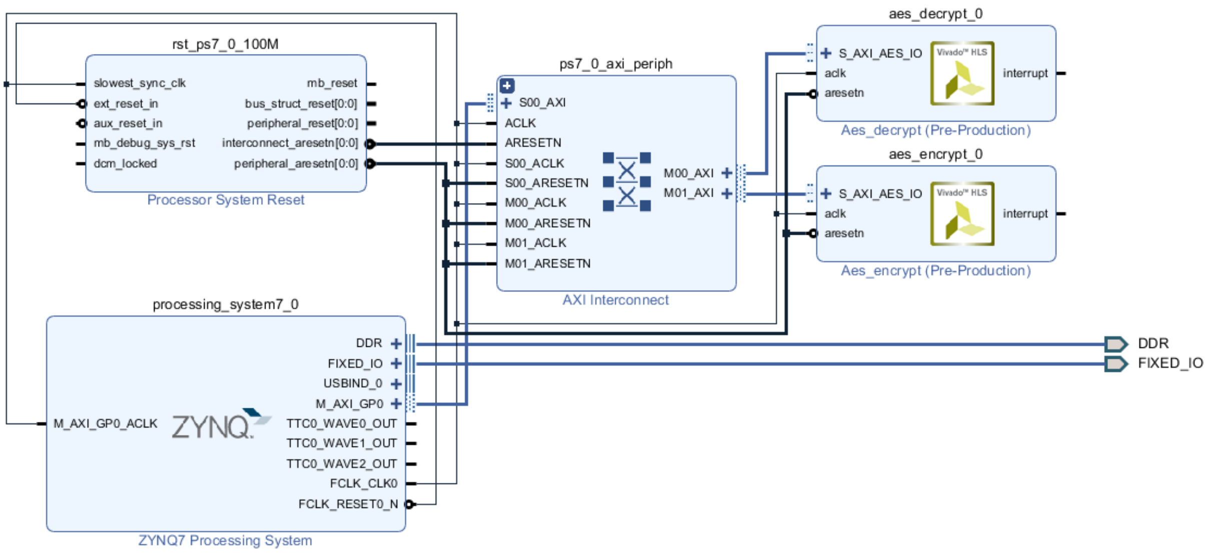
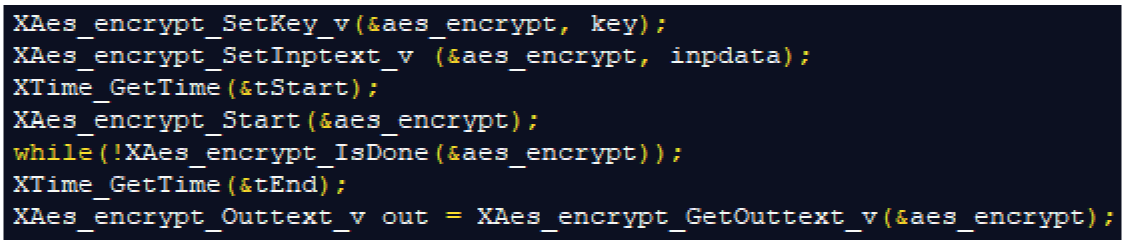


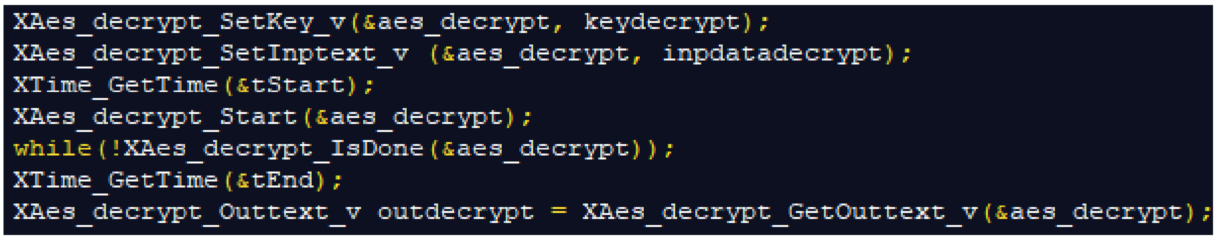
Figure 41: Connecting of the AES modules

The schematic design of the system for testing is presented in the image above. The design again after validation, is synthesized, implemented and a bitstream of it generated. The bitstream is downloaded to the FPGA of the zedboard. At this point the software development kit (SDK) is used to write the software application for testing the encryption and decryption of the data. One 128bit block of data is used to test the encryption with a 128bit block key. If the results match to the testing website and get the initial data from the decryption the IP modules are working correctly.



```
XAes_encrypt_SetKey_v(&aes_encrypt, key);
XAes_encrypt_SetInptext_v (&aes_encrypt, inpdata);
XTime_GetTime(&tStart);
XAes_encrypt_Start(&aes_encrypt);
while(!XAes_encrypt_IsDone(&aes_encrypt));
XTime_GetTime(&tEnd);
XAes_encrypt_Outtext_v out = XAes_encrypt_GetOuttext_v(&aes_encrypt);
```

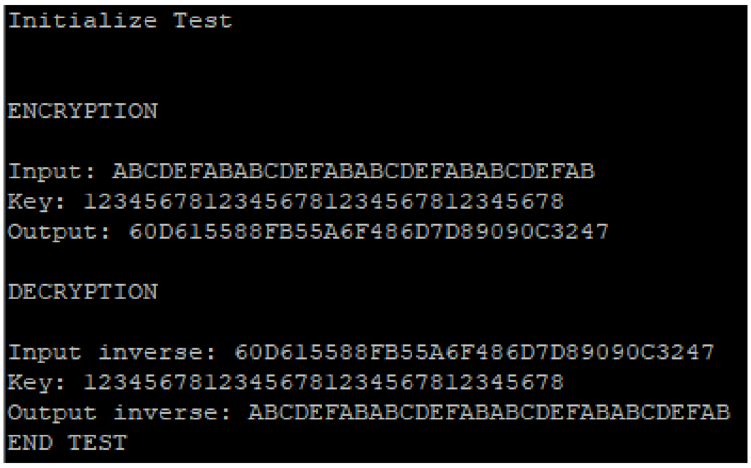
Figure 42: Testing the AES encryption module



```
XAes_decrypt_SetKey_v(&aes_decrypt, keydecrypt);
XAes_decrypt_SetInptext_v (&aes_decrypt, inpdatadecrypt);
XTime_GetTime(&tStart);
XAes_decrypt_Start(&aes_decrypt);
while(!XAes_decrypt_IsDone(&aes_decrypt));
XTime_GetTime(&tEnd);
XAes_decrypt_Outtext_v outdecrypt = XAes_decrypt_GetOuttext_v(&aes_decrypt);
```

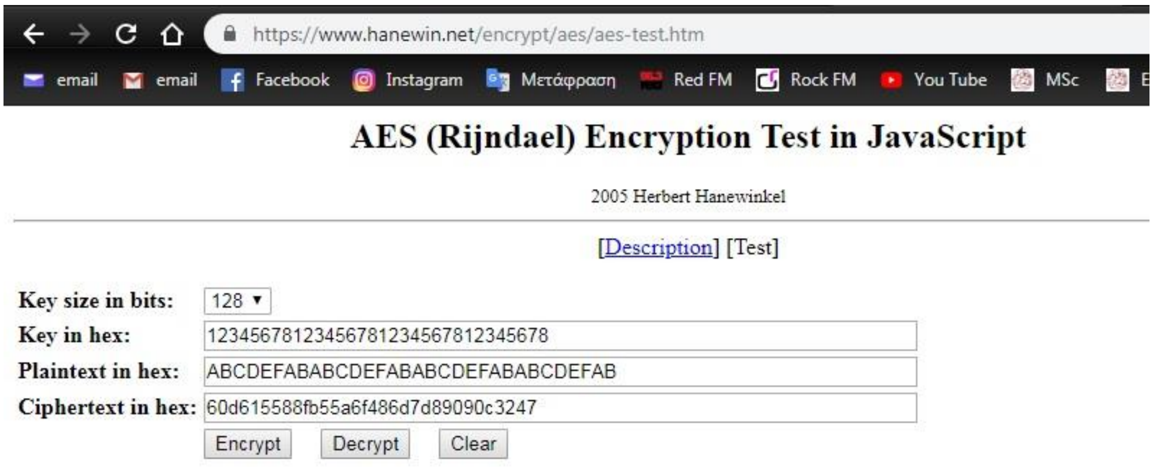
Figure 43: Testing the AES decryption module

Inputs as well as the cryptographic keys in both blocks are given in SKD in Hexadecimal system. The results in the pictures below are printed in hexadecimal system for the ease of presenting. The serial port of the zedboard is used to print the results to a screen with a terminal. The pictures following show the functionality of the modules. The results compared to the real cryptographic results, proof the integrity and authenticity of the blocks developed. Data were successfully encrypted and then returned to initial state as the pictures indicate.



```
Initialize Test


ENCRYPTION

Input: ABCDEFABABCDEFABABCDEFABABCDEFAB
Key: 12345678123456781234567812345678
Output: 60D615588FB55A6F486D7D89090C3247

DECRYPTION

Input inverse: 60D615588FB55A6F486D7D89090C3247
Key: 12345678123456781234567812345678
Output inverse: ABCDEFABABCDEFABABCDEFABABCDEFAB
END TEST
```

Figure 44: Encryption and Decryption results



https://www.hanewin.net/encrypt/aes/aes-test.htm

email email Facebook Instagram Μετάφραση Red FM Rock FM You Tube MSc

AES (Rijndael) Encryption Test in JavaScript

2005 Herbert Hanewinkel

[Description] [Test]

Key size in bits: 128

Key in hex: 12345678123456781234567812345678

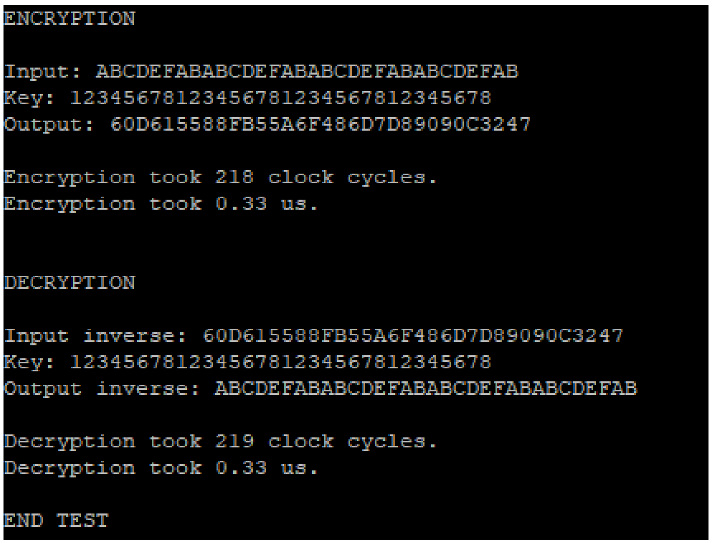
Plaintext in hex: ABCDEFABABCDEFABABCDEFABABCDEFAB

Ciphertext in hex: 60d615588fb55a6f486d7d89090c3247

Encrypt Decrypt Clear

Figure 45: Proof of Operation

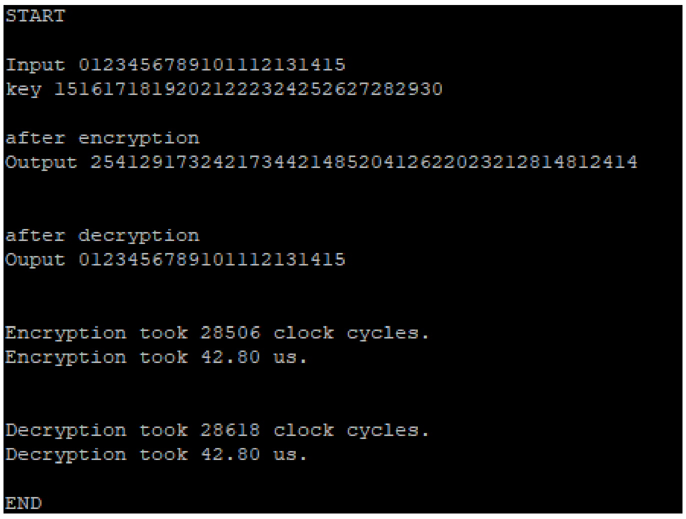
The next step after the confirmation of the functionality of the two blocks is to measure the performance and execution time of each block. The use of the ZYNQ global timer provides the clock cycles used for the encryption and decryption of 128bit block of data. When dividing clock cycles of execution to the ZYNQs frequency, the quotient shows the time needed to execute every operation. Of course, the highest the frequency, the fastest the execution. Measured and tested on the zedboard CPU running maximum 667 MHz the following results came up. It took 218 clock cycles to encrypt a block of 128-bit data and 219 to decrypt them at 0.33 microseconds time of execution. To understand how fast the results came up, it has to be considered that it takes 280 clock cycles for the ZYNQ to execute one simple add function of two integers on a bare metal application.



```
ENCRYPTION

Input: ABCDEFABABCDEFABABCDEFABABCDEFAB
Key: 12345678123456781234567812345678
Output: 60D615588FB55A6F486D7D89090C3247

Encryption took 218 clock cycles.
Encryption took 0.33 us.


DECRYPTION

Input inverse: 60D615588FB55A6F486D7D89090C3247
Key: 12345678123456781234567812345678
Output inverse: ABCDEFABABCDEFABABCDEFABABCDEFAB

Decryption took 219 clock cycles.
Decryption took 0.33 us.

END TEST
```

Figure 46: Performance of Encryption and Decryption blocks

The data processing and performance of the IP blocks are tested in field programmable gate array (FPGA) implementation of the blocks. It has to be made clear how much faster the processes of cryptography can be when implemented as hardware modules. To prove this theory the same AES-code is implemented as software application on the same CPU of the zedboard. The results of the measurements are presented in the image below. As a software implementation of the same code, it took 28.506 clock cycles to encrypt the same amount of data. The encryption hardware block operates 130 times faster than the software implementation. Those results were obtained by processing only 128 bits of data for encryption. That been explained, the hardware block can process new data every one clock cycle. The software version of it has to finish with the first data to process new ones. This is only the first proof of how much faster the encryption can be when operating it as hardware models. The more the data the faster it can be in processing it.



```
START

Input 0123456789101112131415
key 15161718192021222324252627282930

after encryption
Output 25412917324217344214852041262202321281481241


after decryption
Ouput 0123456789101112131415


Encryption took 28506 clock cycles.
Encryption took 42.80 us.


Decryption took 28618 clock cycles.
Decryption took 42.80 us.

END
```

Figure 47: Software implementation of AES code

6.2 System Analysis

After testing every block separately and established that everything is operating within the framework in which was designed, the whole system explained in the previous chapters it to be tested as an integrated system. The next stage is to test the full system with the DMA and single and multiple encryption/decryption modules. The system has to be able to generate new encryption keys and new access rights every time it is set to be changed or every time that's required to, by the system manager.

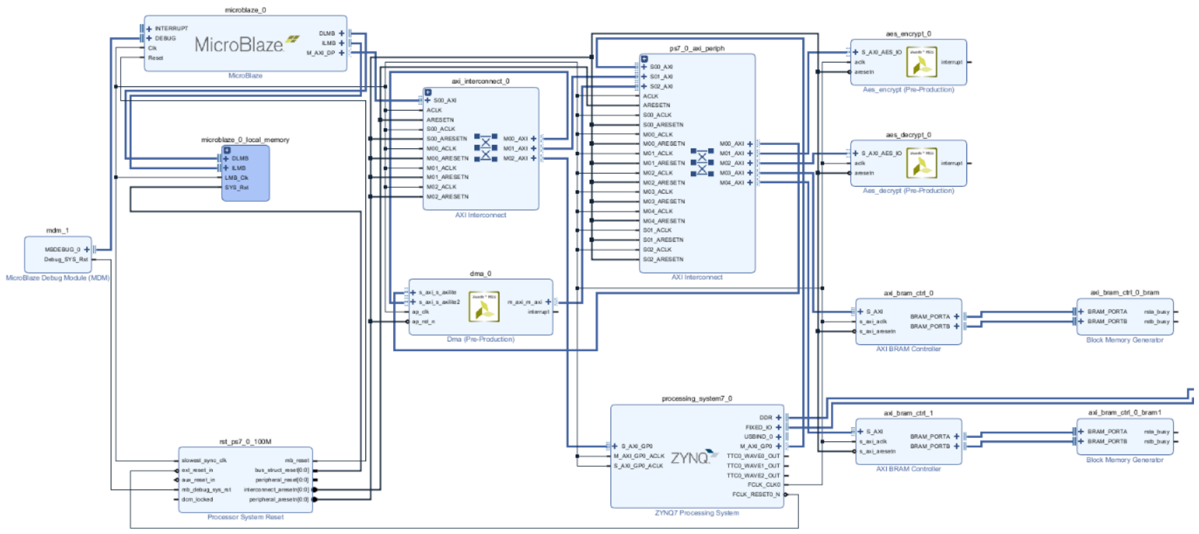
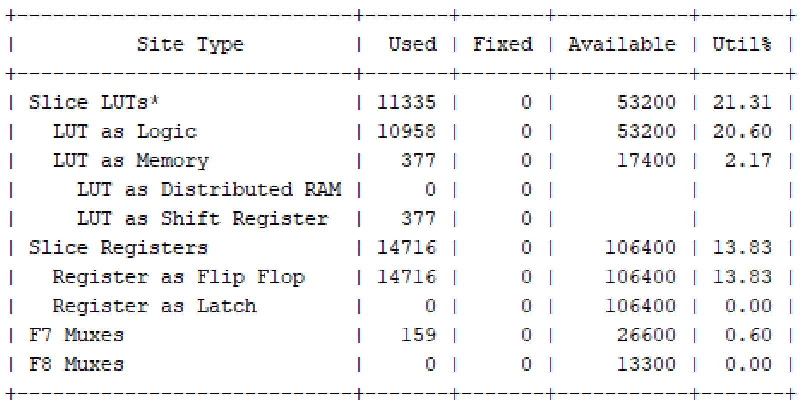


Figure 48: System Design



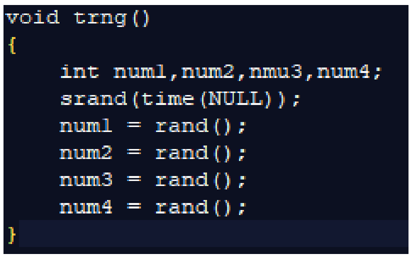
```
+----------------------------+-------+-------+-----------+-------+
|          Site Type         |  Used | Fixed | Available | Util% |
+----------------------------+-------+-------+-----------+-------+
| Slice LUTs*                | 11335 |     0 |     53200 | 21.31 |
|   LUT as Logic             | 10958 |     0 |     53200 | 20.60 |
|   LUT as Memory            |   377 |     0 |     17400 |  2.17 |
|     LUT as Distributed RAM |     0 |     0 |           |       |
|     LUT as Shift Register  |   377 |     0 |           |       |
| Slice Registers            | 14716 |     0 |    106400 | 13.83 |
|   Register as Flip Flop    | 14716 |     0 |    106400 | 13.83 |
|   Register as Latch        |     0 |     0 |    106400 |  0.00 |
| F7 Muxes                   |   159 |     0 |     26600 |  0.60 |
| F8 Muxes                   |     0 |     0 |     13300 |  0.00 |
+----------------------------+-------+-------+-----------+-------+
```

Figure 49: HW Resources

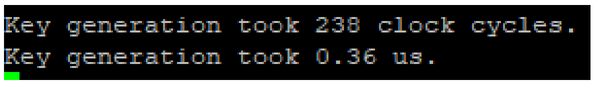
The design above was validated, synthesized, implemented, bitstream was generated and it can be downloaded to the FPGA target device as explained in the previous chapter. At this time the software development kit is use to write applications and test scenarios. The Vivado design suite is used from now on only to implement multiple encryption modules, but that will be explained in the following sections. As a start, applications must be developed for the microblaze and ZYNQ and test the functionality of the design with some test scenarios. As the system responds as it was supposed to, more complicated scenarios can be developed.

The first thing is to write two applications for the microblaze co-processor. The first one is for the boundaries and restrictions that every id has and which peripheral device communicates with which (this application was explained previously in the module evaluation). The second is for key generation and distribution amongst the peripheral devices. A true random number generator (TRNG) application is developed for this purpose. The application creates keys when the session starts and periodically after that. The keys are distributed to all parts of the system securely since only the co-processor has access to them. A TRNG is developed uniquely for this project that creates cryptographic keys by randomly selecting integers as a function of time.

One integer in the system has the size of four bytes (32bits). So, in order to create one 128bit key, the generator has to provide four integers to cover the length of one key. That is the simplest way to look at it, since any type of data is data. One number generator can look like the image below. This function creates four random integers of a total length of 32 bytes (128bits) as a time function and can easily be used as a cryptographic key for this algorithm. It only needs 238 clock cycles to execute, so, there is really not a seriously delay when generating new keys. The distribution of those keys only takes a clock cycle for the co-processor to pass them around. The TRNG used in this project is kept secret because of its complexity and its algorithm will remain hidden for now, but in general shares the same philosophy as the generator below.



```
void trng()
{
    int num1,num2,nmu3,num4;
    srand(time(NULL));
    num1 = rand();
    num2 = rand();
    num3 = rand();
    num4 = rand();
}
```



```
Key generation took 238 clock cycles.
Key generation took 0.36 us.
```

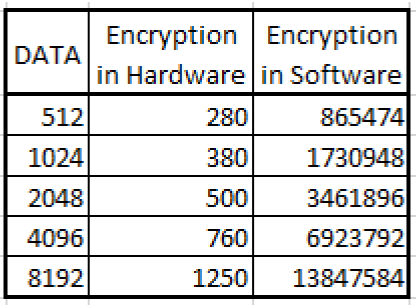
Figure 50: A Random Number Generator

Once the cryptographic keys have been distributed and the restriction arguments set, the main operational application can be developed. To test the functionality of the system one application is running on the ZYNQ processing unit. Several scenarios can be tested on this application depending on when the encryption begins, before the arguments of a DMA transfer are valid or after, maybe encryption inside the DMA module. For every idea or scenario, time execution and performance of the system remains very similar. So, every test could apply to more than one scenario.

Two BRAMs just like in the DMA section are used to test the design. One of them is filled with integers starting from one and the counter increases by one in every slot. The other one left with zeros in order to test the transfer and the encryption. The integers are just for simplicity, those data could have been acquired by a number of peripheral devices (video cameras, audio devices, microphones, etc). Every device that can acquire data has a memory to temporally store it. That is the purpose for those BRAMs.

Suppose that a DMA transfer request occurs in the application, for example private document from a hidden memory to the DDR of the ZYNQ from processing, if the ZYNQ has rights to access data from the hidden memory and the DMA transfer is validated, the data will be transferred after encryption to the ZYNQ CPU. Only the ZYNQ can decrypt those data and no other party of the system who does not have rights. That is accomplished with the keys distributed by the co-processor. When the data are decrypted, they can be processed. The cryptographic key for every id depends on the boundaries and restrictions that have been set. For every id one cryptographic key is generated and is send to both ends than have rights to communicate.

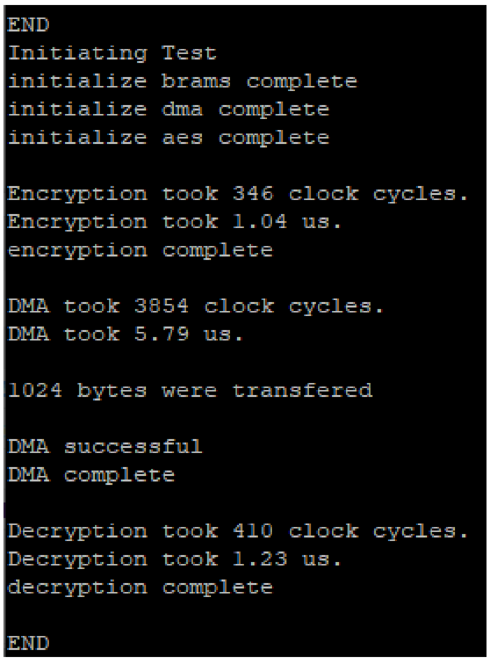
For a test like this, an amount of 1KB of data is used to be transferred and encrypted. We set the arguments in the way that the DMA transfer will be successful in order to test the encryption. 1KB of data means 256 memory slots. It takes about 4000 clock cycles for the DMA to transfer 1KB of data and about 400 clock cycles to encrypt and decrypt them because the hardware encryption modules can accept new data every 1 clock cycle. If the same data were encrypted in software mode it would take 1.730.948 clock cycles to encrypt them making it 4.500 time faster. The more data that are to be encrypted the more it makes sense to add a hardware model for this process. As the data rises the faster, they are encrypted in comparison with the software version of the algorithm. We run the same test for different amount of data and the following results came up.



DATA	Encryption in Hardware	Encryption in Software
512	280	865474
1024	380	1730948
2048	500	3461896
4096	760	6923792
8192	1250	13847584

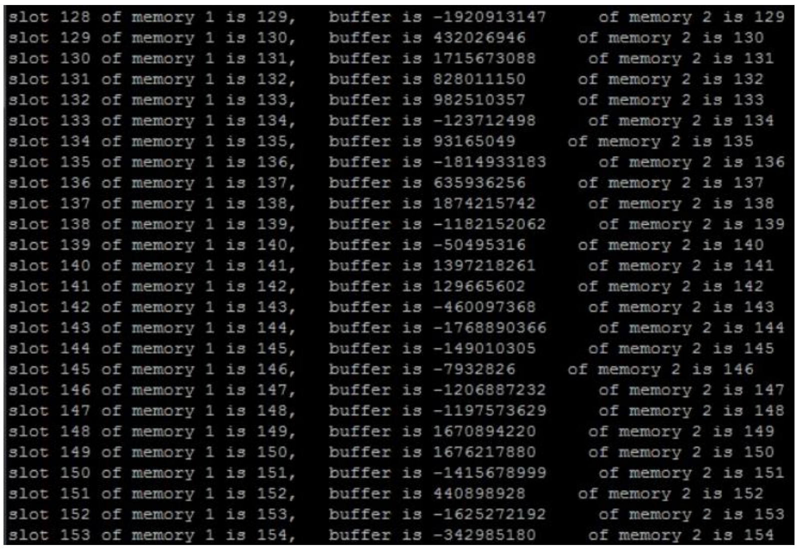
Figure 51: Hardware encryption VS Software encryption (CC)

The serial port of the zedboard is used to print the data and execution time to a terminal display. Putty is used for this purpose. The images bellow shows the results of this test. The buffer in the second image is used to show the encryption of the data and how they are decrypted after they are transferred in the second BRAM.



```
END
Initiating Test
initialize brams complete
initialize dma complete
initialize aes complete

Encryption took 346 clock cycles.
Encryption took 1.04 us.
encryption complete

DMA took 3854 clock cycles.
DMA took 5.79 us.

1024 bytes were transfered

DMA successful
DMA complete

Decryption took 410 clock cycles.
Decryption took 1.23 us.
decryption complete

END
```

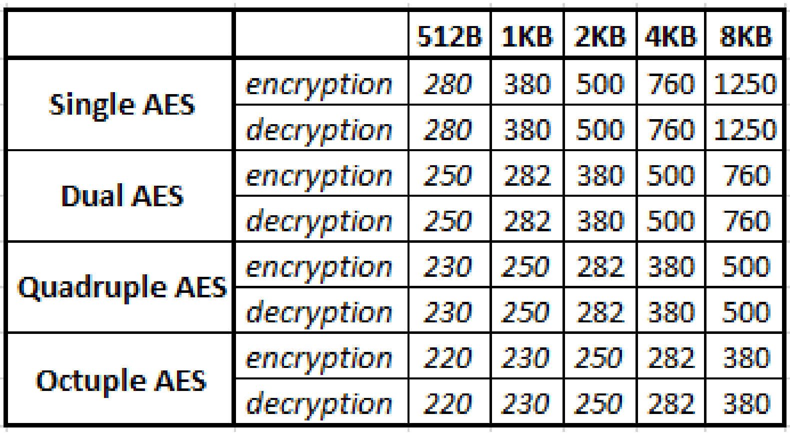
Figure 52: Encryption and DMA transfer of 1KB



```
slot 128 of memory 1 is 129,   buffer is -1920913147    of memory 2 is 129
slot 129 of memory 1 is 130,   buffer is 432026946    of memory 2 is 130
slot 130 of memory 1 is 131,   buffer is 1715673088    of memory 2 is 131
slot 131 of memory 1 is 132,   buffer is 828011150    of memory 2 is 132
slot 132 of memory 1 is 133,   buffer is 982510357    of memory 2 is 133
slot 133 of memory 1 is 134,   buffer is -123712498    of memory 2 is 134
slot 134 of memory 1 is 135,   buffer is 93165049    of memory 2 is 135
slot 135 of memory 1 is 136,   buffer is -1814933183    of memory 2 is 136
slot 136 of memory 1 is 137,   buffer is 635936256    of memory 2 is 137
slot 137 of memory 1 is 138,   buffer is 1874215742    of memory 2 is 138
slot 138 of memory 1 is 139,   buffer is -1182152062    of memory 2 is 139
slot 139 of memory 1 is 140,   buffer is -50495316    of memory 2 is 140
slot 140 of memory 1 is 141,   buffer is 1397218261    of memory 2 is 141
slot 141 of memory 1 is 142,   buffer is 129665602    of memory 2 is 142
slot 142 of memory 1 is 143,   buffer is -460097368    of memory 2 is 143
slot 143 of memory 1 is 144,   buffer is -1768890366    of memory 2 is 144
slot 144 of memory 1 is 145,   buffer is -149010305    of memory 2 is 145
slot 145 of memory 1 is 146,   buffer is -7932826    of memory 2 is 146
slot 146 of memory 1 is 147,   buffer is -1206887232    of memory 2 is 147
slot 147 of memory 1 is 148,   buffer is -1197573629    of memory 2 is 148
slot 148 of memory 1 is 149,   buffer is 1670894220    of memory 2 is 149
slot 149 of memory 1 is 150,   buffer is 1676217880    of memory 2 is 150
slot 150 of memory 1 is 151,   buffer is -1415678999    of memory 2 is 151
slot 151 of memory 1 is 152,   buffer is 440898928    of memory 2 is 152
slot 152 of memory 1 is 153,   buffer is -1625272192    of memory 2 is 153
slot 153 of memory 1 is 154,   buffer is -342985180    of memory 2 is 154
```

Figure 53:Encryption and Decryption

Every result in this test shows that the system operates successful even when the permissions are not granted and the DMA fails. The execution of the operations is proven to be so much faster than the software version of them. That was the idea of this project and can be used in a wide area of medical applications, car automatization applications and basically in every embedded device application were security is important. The next step is to connect multiple encryption and decryption modules and measure performance as well as different amount of data. The following table shows performance measurements on different scenarios.



		512B	1KB	2KB	4KB	8KB
Single AES	*encryption*	*280*	380	500	760	1250
	decryption	*280*	380	500	760	1250
Dual AES	*encryption*	*250*	282	380	500	760
	decryption	*250*	282	380	500	760
Quadruple AES	*encryption*	*230*	*250*	282	380	500
	decryption	*230*	*250*	282	380	500
Octuple AES	*encryption*	*220*	*230*	*250*	282	380
	decryption	*220*	*230*	*250*	282	380

Figure 54:Performance with multiple modules (clock cycles)

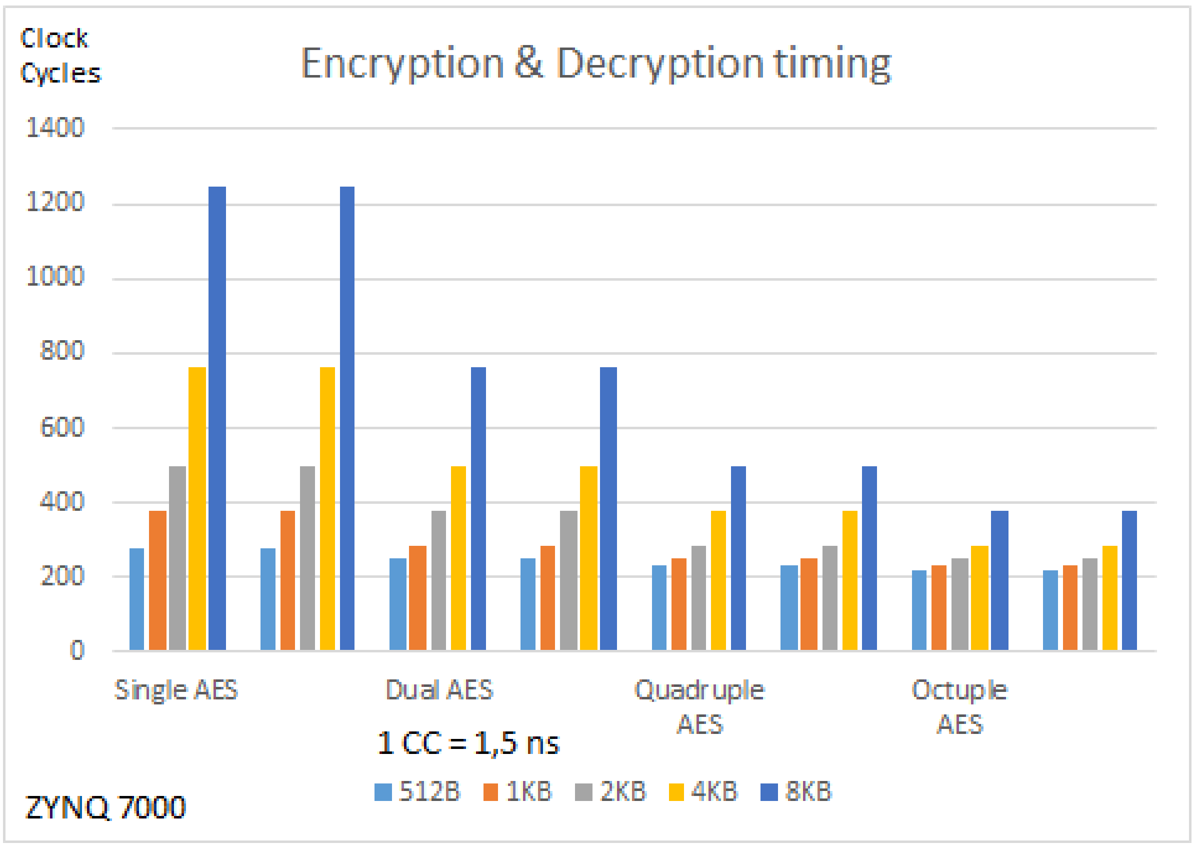


Figure 55: Performance Graph

From the results above it can be made clear that multiple AES encryption hardware blocks make sense when large amount of data need to be encrypted since such blocks use a large amount of resources and the hardware cost rises exponentially. For small amount of data single AES will do just fine.

6.3 System Comparison

The design was targeted to achieve maximum speed in hardware mode, so it lacks flexibility. After a total system analysis, the system can be compared with other systems from related work in the past. In compare to Santosh et al. (2017) this project achieved better performance. They needed 12.300 clock cycles for a software implementation of 128bit key AES and 1032 clock cycles for their hardware implementation. This project achieved it in 280 clock cycles. Mihai et al. (2006) has better performance with 100 clock cycles on 128bit AES encryption to a block of 16bytes of data with a cryptographic accelerator.

Paillier & Verbauwhede (2007) achieved a 15.3 Gbps from their implementation of the AES in GCM mode on a Virtex-4 FPGA under a clock rate of 120 Mhz. In 2011, the implementation of the AES algorithm by Soliman et al. (2011) reached 74 Gbps on a Virtex-5 FPGA under the clock rate of 557 Mhz. In 2013 43. Biglari and Qasemi et al. (2013) design a high-performance AES system that reached a throughput of 12.8 Gbps. In 2016, Smekal et al. (2016) described the AES implementation on Virtex-7 that achieved a 5.1 Gbps throughput under a clock rate of 100 Mhz. Marghescu et al. (2014) developed one complete AES-256 block processing that computed within 180 ns, working at a 100MHz. 47.Gaspar et al. (2012) with the NIOS II-based system achieved an overall throughput of 25.1 Mb/s, the MicroBlaze-based system achieved 18.4 Mb/s and the Cortex M1 system achieved 12.2 Mb/s. Rakanovic et al. (2016) achieved 914 Mb/s throughput with AES256 at 100 MHz frequency.

By analyzing the design, we come to a conclusion that offers security as well as great speed performance. It can be used in a wide area of applications including critical applications where response of the system is a great deal. By using interrupt controllers and prioritizing the IDs, the system gives the opportunity for some devices to gain priority in requesting a DMA transfer or even break a DMA transfer at the time of the request due to the criticality of the operation. The DMA transfer that was interrupted will continue from the point of interrupt after the most critical operation is finished.

6.4 Applications

A secure embedded system should provide security and the cryptographic features such as confidentiality, authenticity, and integrity as well as speed when operating on critical applications. Establishing that the system operates upon those principles, can be set on a large area of embedded device applications. Applications include Automotive industry, Healthcare applications, Telecommunication, Entertainment and multimedia, Robotics, Computer

Networking and basically every system connected to the Internet of Things (IoT). As the IoT expands the devices that connected are vulnerable to various attacks, so systems like the one designed will always be a necessity.

From detecting rash driving on highways to street light control and signal control system with vehicle tracking. Home automation systems with temperature control and smart home management is a fast-growing area for embedded devices. Automatic wireless health monitoring system for patients in medical systems. Automotive, railways and aircraft electronics, military applications, authentication systems, consumer electronics and fabrication equipment, smart buildings and robotics are areas of application for modern embedded systems. The system of this project can provide safe functionality in these areas enforcing the security on critical security parameters with a considerably high performance and low hardware cost.

7. Conclusion and Future work

In this thesis we have designed and developed a system that provides secure access on an embedded system through DMA protection and lightweight cryptography. We developed a DMA module with a priority security sequence, thus, communication between CPU, memories and peripheral devices depend on boundaries and limitations. Those limitations are set by a cryptographic co-processor and generates IDs between devices. Those IDs don't refer to a device, rather than to a communication path between devices for DMA data transfers. In order for the DMA module to execute a DMA transfer, the ID is passed on as argument. The success or failure depends on whether the id is valid or not. If the id is not valid, that means that the two ends do not have permission to exchange data, thus the transfer will fail.

In addition to the security that is provided, two hardware modules were created to perform AES encryption to the data that are transferred through the DMA module. Depending on the IDs the cryptographic co-processor generates a unique cryptographic key for every ID and distributes it to both ends. But doing so, only the receiver can decrypt the data that are transferred. Those keys are generated by a true random number generator that runs in the hidden memory of the cryptographic co-processor at specified times or when requested by the CPU. The design was successfully generated and downloaded to the FPGA of the zedboard for testing in various scenarios and the addition of multiple cryptographic modules.

Evaluation of the system provided very good results in timing and performance with the addition of security, integrity and authenticity. It can have use in a large area of applications and critical applications where real time response is a significant factor. The design is well compared with existing solutions when takin into consideration the hardware cost and the provided performance.

Future extensions can include real system operations with real time needing applications in every possible field that was described in previous chapters. Testing the DMA from this project with possible different cryptographic algorithms and multiple DMA modules, maybe could provide better security. Some variations of the AES algorithm with the use of multiple SBoxes, maybe the use of hardware and software co-operation could provide better results in various tests. Trying to reduce the latency even more could provide better performance in the system. Parallel computing with various cryptographic algorithms on a Network on Chip (NoC) would be a serious improvement in embedded systems security.

Whether we provide better performance, we lose resources and the cost rises. If we provide better security, we lose in speed. If we provide the best in everything, the cost or the hardware size or even the power consumption will be a great restriction. The best idea is trying to provide a balanced solution, using as minimum as possible resources with the best possible outcome, as this project successfully provided.

References

1. Bokhari, M. & Hassan, Shabbir. (2018). A Comparative Study on Lightweight Cryptography. 10.1007/978-981-10-8536-9_8.
2. Wang, Long & Zhao, Hui & Bai, Guoqiang. (2007). A cost-Efficient Implementation of Public-key Cryptography on Embedded Systems. 194 - 197. 10.1109/EDST.2007.4289808.
3. Salman, Ahmad & Diehl, William & Kaps, Jens-Peter. (2017). A light-weight hardware/software co-design for pairing-based cryptography with low power and energy consumption. 235-238. 10.1109/FPT.2017.8280149.
4. Lu, Ronghua & Han, Jun & Zeng, Xiaoyang & Li, Qing & Mai, Lang & Zhao, Jia. (2008). A low-cost cryptographic processor for security embedded system. 113-114. 10.1109/ASPDAC.2008.4483921.
5. Francesc Fons, Mariano Fons, Paul Olivier, André Weimerskirch "A Modular, Reconfigurable and Updateable Embedded Cyber Security Hardware Solution for Automotive", embedded world 2017
6. Zhang, Weilong & Yuan, Yuxiang & Liu, Yang & Zhang, Yapeng & Jiang, Xueping. (2014). A security coprocessor embedded system-on-chip architecture for smart metering, control and communication in power grid. 1-3. 10.1109/ICSICT.2014.7021162.
7. Kanuparthi, Arun & Karri, Ramesh & Ormazabal, Gaston & Addepalli, Sateesh. (2012). A Survey of Microarchitecture Support for Embedded Processor Security. Proceedings - 2012 IEEE Computer Society Annual Symposium on VLSI, ISVLSI 2012. 368-373. 10.1109/ISVLSI.2012.64.
8. 8. Yang, Xiaokun. (2017). An Advanced Bus Architecture for AES-Encrypted High-Performance Embedded Systems
9. 9. Farahmand, Farnoud & Homsirikamol, Ekawat & Gaj, Kris. (2016). A Zynq-based testbed for the experimental benchmarking of algorithms competing in cryptographic contests. 1-7. 10.1109/ReConFig.2016.7857148
10. Hasegawa, Y. & Abe, S. & Matsutani, H. & Amano, H. & Anjo, K. & Awashima, T.. (2006). An adaptive cryptographic accelerator for IPsec on dynamically reconfigurable processor. 2005. 163 - 170. 10.1109/FPT.2005.1568541.
11. Xinyu, li & Omar, Hammami. (2009). An Automatic Design Flow for Data Parallel and Pipelined Signal Processing Applications on Embedded Multiprocessor with NoC: Application to Cryptography. International Journal of Reconfigurable Computing. 2009. 10.1155/2009/631490.
12. Yuan, Hang & Chen, Hongyi & Bai, Guoqiang. (2004). An improved DMA controller for high speed data transfer in MPU based SOC. 1372 - 1375 vol.2. 10.1109/ICSICT.2004.1436811.
13. Conti, Francesco & Schilling, Robert & Schiavone, Pasquale & Pullini, Antonio & Rossi, Davide & Gürkaynak, Frank & Muehlberghuber, Michael & Gautschi, Michael & Loi, Igor & Haugou, Germain & Mangard, Stefan & Benini, Luca. (2016). An IoT Endpoint System-on-Chip for Secure and Energy-Efficient Near-Sensor Analytics. IEEE Transactions on Circuits and Systems I: Regular Papers. PP. 10.1109/TCSI.2017.2698019.
14. Shinagawa, Takahiro & Eiraku, Hideki & Tanimoto, Kouichi & Omote, Kazumasa & Hasegawa, Shoichi & Horie, Takashi & Hirano, Manabu & Kourai, Kenichi & Oyama, Yoshihiro & Kawai, Eiji & Kono, Kenji & Chiba, Shigeru & Shinjo, Yasushi & Kato, Kazuhiko. (2009). Bit visor: A thin hypervisor for enforcing i/o device security. Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE'09. 121-130. 10.1145/1508293.1508311.
15. David G. Conroy, Timothy J. Millet, Michael J. Smith, Joshua P. de Cesare. (2009). CENTRAL DMA WITH ARBITRARY PROCESSING FUNCTIONS. Apple Inc., Cupertino.
16. Reshma Lal , Steven B . McGowan, Siddhartha Chhabra , Gideon Gerzon , Bin Xing , Hillsboro , Pradeep M . Pappachan ,Reouven Elbaz. (2019). CRYPTOGRAPHIC PROTECTION OF I/O DATA FOR DMA CAPABLE I /O CONTROLLERS. Intel Corporation , Santa Clara , CA

17. David A . Kaplan , Thomas Roy Woller, Ronald Perez. (2017). CRYPTOGRAPHIC PROTECTION OF INFORMATION IN A PROCESSING SYSTEM. Advanced Micro Devices , Inc ., Sunnyvale , CA
18. Sen, S. & Hossain, S.I. & Chowdhuri, D.R. & Chaudhuri, P.P.. (2003). Cryptosystem designed for embedded system security. 271- 276. 10.1109/ICVD.2003.1183149.
19. Yoshiyuki Nakai, Koichi Sumida, Takao Yamanouchi, Yohichi Shimazawa. (2009). DATA PROCESSING APPARATUS FOR SELECTING ETHER A PIO DATA TRANSFER METHOD OR A DMA DATA TRANSFER METHOD, Sharp Kabushiki Kaisha, Osaka 2009
20. Charles A. Boone, Robert F. Pfeifer. (1982). DATA SECURITY MODULE. Motorola Inc., Schaumburg, 1982
21. Durga, G.V. & Islam, S. & Sachid, Angada & Meera, P.. (2006). Design and Implementation of a Co-Processor for Providing Data Protection in Embedded Systems. 446- 449. 10.1109/INDCON.2005.1590209
22. Ahmed, Altaf & Aljumah, Abdullah & Ahmad, M. (2019). Design and Implementation of a Direct Memory Access Controller for Embedded Applications. International Journal of Technology. 10. 309. 10.14716/ijtech.v10i2.795.
23. MIHAI TOGAN, ADRIAN FLOAREA, GIGI BUDARIU. (2010). DESIGN AND IMPLEMENTATION OF CRYPTOGRAPHIC MODULES ON FPGAs. Computer Science Department, Military Technical Academy, George Coşbuc Blvd. 81-83, Bucharest, ROMANIA
24. 24. Hai Yan, Zhijie Jerry Shi, and Yunsi Fei. (2009). Efficient Implementation of Elliptic Curve Cryptography on DSP for Underwater Sensor Networks. Department of Electrical and Computer Engineering, University of Connecticut, Storrs
25. Unterluggauer, Thomas & Wenger, Erich. (2014). Efficient Pairings and ECC for Embedded Systems. 298-315. 10.1007/978-3-662-44709-3_17.
26. Ukil, Arijit & Sen, Jaydip & Koilakonda, Sripad. (2011). Embedded Security for Internet of Things. 1 - 6. 10.1109/NCETACS.2011.5751382.
27. Wolff, Francis & Papachristou, Ch & Weyer, Daniel & Clay, William. (2010). Embedded system protection from software corruption. 2010 NASA/ESA Conference on Adaptive Hardware and Systems, AHS 2010. 223 - 229. 10.1109/AHS.2010.5546254.
28. Upadhyay, Aastik & Dhapola, Abhimanyu. (2015). Embedded Systems and its Application in Medical Field. 10.13140/2.1.1299.1528.
29. Papp, Dorottya & Ma, Zhendong & Buttyan, Levente. (2015). Embedded systems security: Threats, vulnerabilities, and attack taxonomy. 145-152. 10.1109/PST.2015.7232966.
30. Parameswaran, Sri & Wolf, Tilman. (2008). Embedded systems security—an overview. Design Autom. for Emb. Sys.. 12. 173-183. 10.1007/s10617-008-9027-x.
31. BENHADDAD Omar Hocine, SAOUDI Mohamed, DROUICHE Amine, RABIAI Mohamed, ALLAILOU Boufeldja. (2019). Hardware Acceleration of AES Cryptographic Algorithm for IPSec. Malaysian Journal of Computing and Applied Mathematics 2019, Vol 2(2): 1-7
32. Wenger, Erich. (2013). Hardware Architectures for MSP430-Based Wireless Sensor Nodes Performing Elliptic Curve Cryptography. 7954. 290-306. 10.1007/978-3-642-38980-1_18.
33. Lukacs, Sandor & Lutas, Andrei & Lutas, Dan & Sebestyen, Gheorghe. (2014). Hardware virtualization-based security solution for embedded systems. 1-6. 10.1109/AQTR.2014.6857879.
34. Koschuch, Manuel & Lechner, Joachim & Weitzer, Andreas & Großschädl, Johann & Szekely, Alexander & Tillich, Stefan & Wolkerstorfer, Johannes. (2006). Hardware/Software Co-design of Elliptic Curve Cryptography on an 8051 Microcontroller. 4249. 430-444. 10.1007/11894063_34.
35. Sharif, Malik & Shahid, Rabia & Gaj, Kris & Rogawski, Marcin. (2016). Hardware-software codesign of RSA for optimal performance vs. flexibility trade-off. 1-4. 10.1109/FPL.2016.7577368.
36. Seuschek, Hermann & Khurana, Piyush & Sigl, Georg. (2015). HiPeC — High Performance Cryptographic Service for Heterogeneous Network-on-Chip Systems. IFAC-PapersOnLine. 48. 31-36. 10.1016/j.ifacol.2015.07.003.

37. Ken, Cai & Xiaoying, Liang. (2010). Implementation of RSA Algorithm Using SOPC Technology. 10.1109/EBISS.2010.5473562.
38. Anwar, Hassan & Daneshtalab, Masoud & Ebrahimi, Masoumeh & Ramirez, Marco & Plosila, Juha & Tenhunen, Hannu. (2014). Integration of AES on Heterogeneous Many-Core System. 424-427. 10.1109/PDP.2014.86.
39. Rakanovic, Damjan & Struharik, Rastislav. (2016). IP core for AES256 and TDES algorithms with AXI interface. 1-4. 10.1109/TELFOR.2016.7818860.
40. Santosh Ghosh, Rafael Misoczki, Li Zhao and Manoj R Sastry. (2017) Lightweight Block Cipher Circuits for Automotive and IoTSensor Devices. HASP '17: Proceedings of the Hardware and Architectural Support for Security and PrivacyJune 2017 Article No.: 5 Pages 1–7https://doi.org/10.1145/3092627.3092632
41. Kalamkar, Gaurav & Gotkhindikar, Ajey & Suryawanshi, A.. (2018). Literature survey on memory level attacks in Automotive Embedded Systems. 1-10. 10.1109/ICCUBEA.2018.8697376. IJARIIE-ISSN(O)-2395-4396.
42. Kalamkar, Gaurav & Gotkhindikar, Ajey & Suryawanshi, A.. (2018). Low-Level Memory Attacks on Automotive Embedded Systems. 1-5. 10.1109/ICCUBEA.2018.8697376.
43. Biglari, Mehrdad & Qasemi, Ehsan & Pourmohseni, Behnaz. (2013). Maestro: A high performance AES encryption/decryption system. 145-148. 10.1109/CADS.2013.6714255.
44. Sachin Gupta & Lakshmi Natarajan. (2010). Optimizing Embedded Applications using DMA. Cypress Semiconductor Corp. Published in EE Times Design (http://www.eetimes.com)
45. Mühlbach, Sascha & Wallner, Sebastian. (2008). Secure communication in microcomputer bus systems for embedded devices. Journal of Systems Architecture. 54. 1065-1076. 10.1016/j.sysarc.2008.04.003.
46. Marghescu, Andrei & Svasta, Paul. (2014). Secure communication protocol using embedded devices based on FPGA. 1-4. 10.1109/ESTC.2014.6962858.
47. Gaspar, Lubos & Fischer, Viktor & Bossuet, Lilian & Fouquet, Robert. (2012). Secure Extension of FPGA General Purpose Processors for Symmetric Key Cryptography with Partial Reconfiguration Capabilities. ACM Transactions on Reconfigurable Technology and Systems. 5. 10.1145/2362374.2362380.
48. Areno, Matthew & Plusquellic, J.. (2013). Secure mobile authentication and device association with enhanced cryptographic engines. 1-8. 10.1109/PRISMS.2013.6927180.
49. Schwarz, Oliver & Gehrmann, Christian. (2012). Securing DMA through virtualization. 1-6. 10.1109/CompEng.2012.6242958.
50. Kumaki, Takeshi & Koide, Tetsushi & Mattausch, H.J. & Tagami, Masaharu & Ishizaki, Masakatsu. (2011). Software-Based Parallel Cryptographic Solution with Massive-Parallel Memory-Embedded SIMD Matrix Architecture for Data-Storage Systems. IEICE Transactions on Information and Systems. E94-D. 1742-1754. 10.1587/transinf.E94.D.1742.
51. Studnia, Ivan & Nicomette, Vincent & Alata, Eric & Deswarte, Yves & Kaaniche, Mohamed & Laarouchi, Youssef. (2013). A Survey of Security Threats and Protection Mechanisms in Embedded Automotive Networks. Proceedings of the International Conference on Dependable Systems and Networks. 1-12. 10.1109/DSNW.2013.6615528.
52. Malcolm, Ronald Smith & Kshitiz, Vadera & Mark, Philip & Zagrodney, Kevin & Ka Wai Ng & Afshin, Rezayee. (2015). Systems and methods for secure processing with embedded cryptographic unit. Square, Inc., San Francisco, CA (US)
53. Stewin, Patrick & Bystrov, Iurii. (2012). Understanding DMA Malware. 10.1007/978-3-642-37300-8_2.
54. Wikipedia – Embedded Systems. (https://en.wikipedia.org/wiki/Embedded_system)
55. Wikipedia – Embedded Systems Security (https://en.wikipedia.org/wiki/Embedded_system)
56. Wikipedia – Cryptography (https://en.wikipedia.org/wiki/Cryptography)
57. A Guide to Using Direct Memory Access in Embedded Systems, Pebble Bay, Embedded Software Development. (https://www.pebblebay.com/direct-memory-access-embedded-systems/)

58. George Kornaros, Othon Tomoutzoglou and Marcello Coppola, “Hardware-assisted Security in Electronic Control Units Utilizing One-Time-Programmable Network-on-Chip and Firewalls”, IEEE Micro, Volume: 38, Issue: 5, Sep./Oct. 2018, pp. 63-74, 2018 DOI: https://ieeexplore.ieee.org/abstract/document/8474944
59. G. Kornaros (Editor) Multi-Core Embedded Systems, CRC Press/Taylor & Francis Group, 07-April-2010, ISBN: 978-1-4398-1161-0
60. G. Kornaros, D. Bakoyiannis, O. Tomoutzoglou, M. Coppola and G. Gherardi, "TrustNet: Ensuring Normal-world and Trusted-world CAN-bus Networking," 2019 IEEE International Conference on Communications, Control, and Computing Technologies for Smart Grids (SmartGridComm), Beijing, China, 2019, pp. 1-6. doi: 10.1109/SmartGridComm.2019.8909715
61. Dimitris Mbakoyiannis, Othon Tomoutzoglou, and George Kornaros, “Secure Over-the-air Firmware Updating for Automotive Electronic Control Units”, Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing (SAC '19), pp. 174—181, Limassol, Cyprus, 2019, doi: 10.1145/3297280.3297299, url: http://doi.acm.org/10.1145/3297280.3297299
62. Georgios Kornaros and Marcello Coppola, “Enabling Efficient Job Dispatching in Accelerator-extended Heterogeneous Systems with Unified Address Space”, Procs of 30th International Symposium on Computer Architecture and High-Performance Computing, SBAC-PAD 2018, September 24-27, 2018, doi: 10.1109/CAHPC.2018.8645945, URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8645945&isnumber=8645847 (Acc. Rate: 27%)
63. George Kornaros and Svoronos Leivadaros, "Securing Dynamic Firmware Updates of Mixed-Critical Applications", 3rd IEEE International Conference on Cybernetics (CYBCONF), 2017, pp. 1-7, doi:10.1109/CYBConf.2017.7985807
64. George Kornaros, Ernest Wozniak, Oliver Horst, Nora Koch, Christian Prehofer, Alvise Rigo, Marcello Coppola, "Secure and Trusted Open CPS Platforms", in book "Handbook of Research on Solutions for Cyber-Physical Systems Ubiquity", Editors: Norbert Druml, Andreas Genser, Armin Krieg, Manuel Menghin and Andrea Hoeller, IGI Global book series Advances in Systems Analysis, Software Engineering, and High Performance Computing (ASASEHPC) (ISSN: 2327-3453; eISSN: 2327-3461), 2017
65. Christian Prehofer, Oliver Horst, Riccardo Dodi, Arjan Geven, George Kornaros, Eleonora Montanari, Michele Paolino, “Towards Trusted Apps platforms for open CPS”, 3rd International Workshop on Emerging Ideas and Trends in Engineering of Cyber-Physical Systems, (EITEC, CPSWeek0, 2016, Vienna, Austria, April 11, 2016, pp. 23-28, DOI: 10.1109/EITEC.2016.7503692
66. George Kornaros, Ioannis Christoforakis, Othon Tomoutzoglou, Dimitrios Bakoyiannis, Kallia Vazakopoulou, Miltos Grammatikakis, Antonis Papagrigoriou, “Hardware Support for Cost-Effective System-Level Protection in Multi-core SoCs.”, 2015 Euromicro Conference on Digital System Design, DSD 2015, Madeira, Portugal, August 26-28, 2015, pp. 41-48, DOI: 10.1109/DSD.2015.65
67. Wikipedia - Information Security (https://en.wikipedia.org/wiki/Information_security)
68. Wikipedia - Direct Memory Access (https://en.wikipedia.org/wiki/Direct_memory_access)
69. Wikipedia - DMA Attack (https://en.wikipedia.org/wiki/DMA_attack)
70. Wikipedia - Advanced Encryption Standard (https://en.wikipedia.org/wiki/Advanced_Encryption_Standard)
71. Wikipedia – True Random Number Generator (https://en.wikipedia.org/wiki/Hardware_random_number_generator)
72. Wikipedia - Xilinx Vivado (https://en.wikipedia.org/wiki/Xilinx_Vivado)
73. Paillier, P & Ingrid Verbauwhede. (2007). Multi-gigabit GCM-AES Architecture Optimized for FPGAs. In Cryptographic Hardware and Embedded Systems - CHES. Springer Berlin Heidelberg, Berlin, Heidelberg.

74. Soliman, Mostafa I & Ghada Y Abozaid. (2011). FPGA implementation and performance evaluation of a high throughput crypto co-processor. J. Parallel and Distrib. Comput. 71, 8, 1075–1084.
75. Smekal, D, Jakub Frolka, and Jan Hajny. (2016). Acceleration of AES Encryption Algorithm Using Field Programmable Gate Arrays. 14th IFAC Conference on Programmable Devices and Embedded Systems, 49, 25 (2016), 384 – 389.
76. Bossuet, L., Grand, M., Gaspar, L., Fischer, V., and Gogniat, G. (2013). Architectures of Flexible Symmetric Key Crypto Engines & Mdash; a Survey : From Hardware Coprocessor to Multi-crypto-processor System on Chip. ACM Comput. Surv., 45(4).
77. Tim Good, Mohammed Benaissa, "AES on FPGA from the Fastest to the Smallest," in Cryptographic Hardware and Embedded Systems-CHES 2005, Berlin Heidelberg: Springer-Verlang, 2005, pp 427-440.
78. Hoang Trang, Nguyen Van Loi, "An efficient FPGA implementation of the Advanced Encryption Standard algorithm," in 2012 IEEE RIVF, pp. 55-59
79. Sever R, Ismailglu AN, Tekmen YC, Askar M, Okcan B, "A high speed FPGA implementation of the Rijndael algorithm," Euromicro Symposium on Digital System Design, 2004, pp. 358–362.
80. Wang SS, Ni WS, "An efficient FPGA implementation of advanced encryption standard algorithm", International Symposium on Circuits and Systems, Vol. 2, pp. 597-600, 2004.
81. Wikipedia - Advanced eXtensible Interface (https://en.wikipedia.org/wiki/Advanced_eXtensible_Interface)